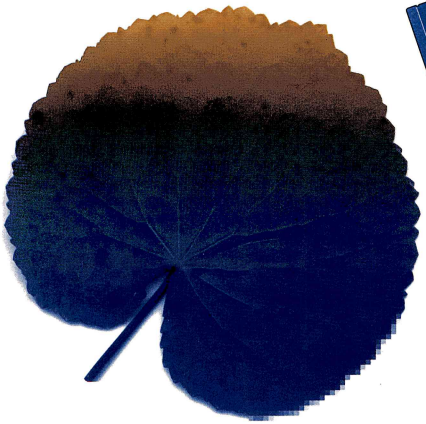
Haselier • Fahnenstich (Hrsg.)

Ulrich Cuber
Heino Wenzel

Das Einmaleins der

Pascal-Programmierung



ADDISON-WESLEY

ECON

Das Einmaleins der Pascal-Programmierung

ECON-Computer-Taschenbuch

Herausgegeben von
rabbitsoft Haselier & Fahnenstich

Ulrich Cuber
Heino Wenzel

Das Einmaleins der Pascal-Programmierung

Addison-Wesley Publishing Company
ECON Taschenbuch Verlag

Cuber, Ulrich; Wenzel, Heino:
Das Einmaleins der Pascal-Programmierung/Ulrich Cuber; Heino Wenzel. [Hrsg.: Rabbitsoft Rainer G. Haselier und Klaus Fahnenstich]. - Bonn; Paris [u.a.]: Addison-Wesley; Düsseldorf: ECON Verlag, 1996
(ECON ; 28086: ECON-Computer-Taschenbuch)
ISBN 3-612-28086-4

Veröffentlicht im ECON Taschenbuch Verlag

Originalausgabe
© 1996 by ECON Verlag GmbH, Düsseldorf

Herausgeber: rabbitsoft Rainer G. Haselier und Klaus Fahnenstich
Lektorat: Markus Heimann
Umschlaggestaltung: Molesch/Niedertubbesing
Layout: Bertram Eckert
Satz: rabbitsoft Rainer G. Haselier und Klaus Fahnenstich
Belichtung: GOFFINTEXT, Aachen
Druck und Bindearbeiten: Ebner, Ulm

Text, Abbildungen und Programme wurden mit größter Sorgfalt erarbeitet. Verlag, Herausgeber und Autoren können jedoch für eventuell verbliebene fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Kein Teil des Buches darf ohne schriftliche Genehmigung der Herausgeber in irgendeiner Form durch Fotokopie, Mikrofilm oder andere Verfahren reproduziert oder in eine für Maschinen, insbesondere Datenverarbeitungsanlagen, verwendbare Sprache übertragen werden. Auch die Rechte der Wiedergabe durch Vortrag, Funk und Fernsehen sind vorbehalten.
Die in diesem Buch erwähnten Software- und Hardwarebezeichnungen sind in den meisten Fällen auch eingetragene Warenzeichen und unterliegen als solche den gesetzlichen Bestimmungen.

ISBN 3-612-28086-4

Inhaltsverzeichnis

Einleitung ***11***

1 Was kann Borland-Pascal alles? 12
2 Was wir davon in diesem Buch behandeln 14

1 ***Grundlagen*** ***19***

1.1 Pascal 20
1.1.1 Was ist ein Pascal-Programm? 20
1.1.2 Beschreibung der Programm-Elemente 22
1.2 Syntax von Pascal 23
1.2.1 Im Programm erlaubte Zeichen 23
1.2.2 Schlüsselworte und Standardanweisungen 24
1.2.3 Namensregeln 25
1.3 Kommentare und Compilerdirektiven 27
1.4 Programme 29
1.4.1 Der Programmaufbau 29
1.4.2 Einsatz externer Bibliotheken 32
1.5 Programmausgaben 33
1.5.1 Nette Grüße 34
1.5.2 Formatierte Ausgabe 36
1.6 Ausblicke 38

2 Weltwissen 39

2.1 Was sind Datentypen? 40
2.1.1 Die Idee der Datentypen 41
2.1.2 Vordefinierte Datentypen 42
2.1.3 Besondere Datentypen 44
2.1.4 Benutzerdefinierte Datentypen 45
2.2 Konstanten und Variablen 46
2.2.1 Literale Konstanten 46
2.2.2 Symbolische Konstanten 48
2.2.3 Variablen und ihre Verwendung 51
2.3 Die Eigenschaften ordinaler Datentypen 54
2.3.1 Behandlung ordinaler Datentypen 54
2.3.2 Zeichen 58
2.3.3 Ganzzahlen 58
2.3.4 Wahrheitswerte 62
2.4 Reale Datentypen 63
2.4.1 Datentypen 63
2.4.2 Fehlerquellen 63
2.5 Zeichenketten 64
2.5.1 Zeichenketten sind nicht gleich Zeichenketten! 64
2.5.2 Der Datentyp string 65
2.5.3 Stringfunktionen 66
2.6 Kleines Intermezzo: Dateneingabe 68
2.6.1 Eingabefunktionen Read() und ReadLn() 69
2.6.2 Einlesen von Zahlen 69
2.6.3 Unterschiede zwischen Read() und ReadLn() 70

3 Mit Pascal können Sie rechnen 71

3.1 Ausdruck, Anweisung, Anweisungsblock 72
3.1.1 Der Ausdruck 72
3.1.2 Die Anweisungen und der Anweisungsblock 74

3.2 Operatoren 76
3.2.1 Arithmetische Operatoren 76
3.2.2 Vergleichsoperatoren 79
3.2.3 Logische (Boolsche) Operatoren 81
3.2.4 Bitweise Operatoren 84
3.3 Regeln für die Typumwandlung 90
3.3.1 Gleich und gleich ... 90
3.3.2 Mit Gewalt geht (fast) alles 91
3.3.3 C/C++ läßt grüßen! 93

4 *Die Programmsteuerung* 95

4.1 Die Darstellung von Programmen 96
4.1.1 Die Elemente der Struktogramme 97
4.1.2 Struktogramm = Programm 103
4.2 Verzweigung und Mehrfachauswahl 104
4.2.1 Die Verzweigung mit if 104
4.2.2 Die Mehrfachauswahl 109
4.3 Schleifen 113
4.3.1 Die kopfgesteuerte while-Schleife 114
4.3.2 Die fußgesteuerte repeat-Schleife 116
4.3.3 Die Zählschleife mit for 118
4.4 Sprünge 120
4.4.1 Das »böse« goto 121
4.4.2 Schleifensteuerung mit break und continue 123
4.4.3 Andere Sprünge 125

5 *Öfter mal was Neues* 127

5.1 Die Typdefinitionen 128
5.1.1 Implizite Typdefinitionen 128
5.1.2 Explizite Typdefinitionen 131
5.1.3 Typkonstanten 135
5.2 Aufzählungen 137
5.3 Unterbereiche 138
5.4 Mengen 140

5.5 Arrays 145
5.5.1 Definition eines Arrays 145
5.5.2 Zugriff auf Arrayelemente 148
5.6 Records und variante Records 151
5.6.1 Records 152
5.6.2 Variante Records 156

6 Dateizugriffe 161

6.1 Einfache Grundoperationen mit Dateien 162
6.1.1 File of ... 162
6.1.2 Textdateien 163
6.1.3 Typisierte Dateien 165
6.1.4 Untypisierte Dateien 168
6.1.5 Lesen und Schreiben einer Datei 169
6.1.6 Eine eigene Fehlerbehandlung 173
6.1.7 Ein abschließendes Beispiel 175
6.1.8 Interessantes aus der Unit System 178
6.2 MS-DOS läßt grüßen 179
6.2.1 Zusätzliche Funktionen in Borland-Pascal 179
6.2.2 Beispiel: Ermittlung der Plattenauslastung 181

7 Teile und Herrsche 185

7.1 Was sind Subroutinen? 186
7.1.1 Funktionen und Prozeduren 187
7.1.2 Bibliotheken 189
7.2 Definition einer Subroutine 192
7.2.1 Allgemeines 192
7.2.2 Der Prozedur- und Funktionskopf 195
7.2.3 Das Innere der Subroutine 196
7.2.4 Aufruf 200
7.3 Variablen und Subroutinen 203
7.3.1 Parameter 203
7.3.2 Globale und lokale Datenobjekte 208

7.4 Die Definition von Units 215
7.4.1 Der Aufbau einer Unit 215
7.4.2 Der Interfaceteil 216
7.4.3 Der Implementationsteil 217
7.4.4 Der Initialisierungsteil 219

8 Dynamische Speicherverwaltung 221

8.1 Der Hauptspeicher 222
8.1.1 Speicherplatz - Adresse - Variable 223
8.1.2 Stack- und Heapgrößen 224
8.1.3 Speicherprobleme 226
8.2 Anlage und Zugriff 227
8.2.1 Anlage der Zeiger 227
8.2.2 Anlage einer typisierten Zeigervariablen 228
8.2.3 Andere Zeigertypen 231
8.2.4 Zugriff auf Zeigerinhalte 232
8.3 Zeigerinitialisierungen 233
8.3.1 Initialisierung mit festen Datenobjekten 233
8.3.2 Dynamische Speicherverwaltung 236
8.4 Zum Abschied 240

A Pascal im Überblick 241

A.1 Compiler-Direktiven 242
A.2 Parameterbefehle 242
A.3 Befehle zur bedingten Compilierung 243
A.4 Reservierte Worte 243
A.5 Standardanweisungen 245

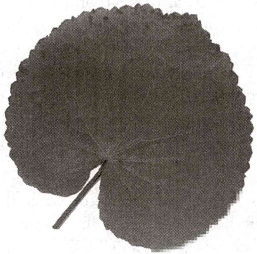
B Die Diskette 247

Index 249

Einleitung

1 Was kann Borland-Pascal alles? *12*

2 Was wir davon in diesem Buch behandeln *14*



Dieses Buch möchte Sie in die Grundlagen der Programmiersprache Pascal einführen. Wir beziehen uns dabei nicht auf die vom Erfinder der Sprache, Niklaus Wirth, gegebene Definition, sondern werden Ihnen den erweiterten Sprachdialekt von Borland Pascal präsentieren. Der Grund dafür liegt darin, daß Borland Pascal in der Version 7.0 der am weitesten verbreitete Pascal-Dialekt ist. Er ist ein direkter Nachfahre des legendären Turbo-Pascal, mit dem auch die Firma Borland ins Leben trat.

1 Was kann Borland-Pascal alles?

Im Gegensatz zum Original-Pascal ist Borland-Pascal um viele Funktionen und Möglichkeiten erweitert worden, die sich im Lauf der Zeit als nützlich herausgestellt haben. Und auch eine Erweiterung in Richtung objektorientierte Programmierung ist vorhanden. Aus den zaghaften Anfängen der Version 6.0 ist eine echte OOP-Sprache geworden. Des weiteren ist Borland Pascal 7.0 die Basis des Delphi-Programmiersystems, welches ebenfalls von Borland entwickelt und vertrieben wird.

Delphi selber ist eine Entwicklungsumgebung, um mit einem auf Pascal 7.0 basierenden Dialekt und der erweiterten Klassenbibliothek OWL schnell und einfach neue Windows-Programme zu erzeugen.

DOS-Programme

Um neben den einfachsten Programmen auch wirklich professionelle Anwendungen »stricken« zu können, hat Borland selber für die Programmierung von DOS-Anwendungen eine Menge Bibliotheksfunktionen beigelegt. Dazu gehören Funktionen, mit denen Sie den Testbildschirm manipulieren oder eine Grafikausgabe vornehmen können. Daneben gibt es Bibliotheken, die auch elementare Hardwarebausteine ansprechen und somit den gesamten PC ohne »Wenn und Aber« unter Ihre Kontrolle bringen.

Im Rahmen der objektorientierten Erweiterungen wurde auch für DOS-Programme eine Objektbibliothek entwickelt – ihr Name ist »Turbo Vision« – mit der es möglich sein soll, schnell und unkom-

pliziert anspruchsvolle textbasierte Benutzerschnittstellen zu entwerfen. Ob das mit dem »unkompliziert« so ganz geklappt hat, möchten wir bezweifeln. Denn die Erstellung eines ereignisgesteuerten Programms mit Mausunterstützung, Menüs und Fenstern ist nun einmal keine simple Angelegenheit. Vor allem, wenn das Betriebssystem fast keine Unterstützung anbietet.

Ein weiterer Aspekt der Handhabung von DOS ist die Möglichkeit, innerhalb von Pascal-Programmen ganze Blöcke in Assembler oder sogar direkt in Maschinensprache einzufügen. Damit kann man zumindest kleinere Programme komplett in Assembler realisieren, ohne dafür die Programme wie Turbo-Assembler nutzen zu müssen.

Windows-Programme

Mit Borland Pascal 7.0 ist es möglich, daß Sie Windows-Programme schreiben. Die Sprache hat zu diesem Zweck einige Erweiterungen erfahren, die sie an C/C++ angleichen sollen, die Sprache, die gemeinhin für die Windows-Programmierung verwendet wird. Alle Werkzeuge und Hilfedateien sind vorhanden, es ist nicht nötig, das SDK von Microsoft extra zu erwerben. Und die Entwicklungsumgebung liegt neben der DOS-Version auch als Windows-Version vor.

Trotzdem müssen wir Sie vor verfrühtem Optimismus warnen: Die Windowsprogrammierung ist auch dann noch kein simples Geschäft!

Objektorientierte Programmierung

Neben der Möglichkeit, »klassische« prozedurale Programme zu schreiben, bietet Ihnen Pascal auch den objektorientierten Weg an. Daher auch der Name »Object Pascal«, den Borland für diese neue Abwandlung von Pascal gewählt hat. Das normale Pascal ist in dieses »Object Pascal« eingebettet und kann genutzt werden, ohne daß Sie sich jemals mit der objektorientierten Programmierung auseinandersetzen müßten. Sollten Sie aber die Absicht haben, sich mit einer OOP-Sprache vertaut zu machen, können Sie diesen Borland-Compiler sofort nutzen.

Portabilität

Sie sehen, die Beschäftigung mit Borland Pascal lohnt sich. Einen Wermutstropfen wollen wir aber nicht verhehlen: Da dieser Sprachdialekt das Produkt einer einzigen Firma ist, die ihren Schwerpunkt der Aktivitäten auf den DOS/Windows-Bereich setzt, werden Sie bei der Übertragung Ihrer Programme auf andere Compiler und/oder andere Betriebssysteme erhebliche Schwierigkeiten haben. Sollten Sie so etwas planen, beschäftigen Sie sich besser mit Progammiersprachen, die plattformübergreifend normiert sind und auch auf verschiedenen Plattformen zur Verfügung stehen. Beipsielsweise mit C oder C++. Ansonsten ist (Object-)Pascal aber ein ausgezeichnetes Werkzeug für die Entwicklung von DOS/Windows-Anwendungen. Und der reine Pascal-Sprachkern ist auch gut geeignet, das Programmieren zu erlernen.

2 Was wir davon in diesem Buch behandeln

Unsere kleine Aufzählung zeigt schon die gigantischen Möglichkeiten und Erweiterungen, die in Borland-Pascal allein für die Schaffung professioneller Programme stecken. Daß es ziemlich sinnlos ist, diese Themen mit Gewalt in ein Taschenbuch zu stopfen, ist auch offensichtlich.

So würde es den Rahmen des Buches sprengen, wenn wir auch nur ansatzweise versuchen würden, die Windowsprogrammierung mit Pascal zu beschreiben. Geschweige denn, wenn wir »das große Buch zu Borland-Pascal« machen wollten. Diese Themen seien – genug Publikumsinteresse vorausgesetzt – späteren Büchern vorbehalten.

Und so standen wir denn vor dem Dilemma: »Was soll rein?« Wir haben uns dafür entschieden, einen Einführungsband für den Programmieranfänger zu schreiben. Wenn Sie also »Profi« sind, dann ist dieses Buch wirklich nichts für Sie!

Die Kapitelinhalte

Daher nun die Liste der Kapitel mit ihren Inhalten. Da man am besten lernt, wenn man mit der Programmiersprache experimentiert, haben wir wieder viele kleine Beispiele benutzt, um Sie schrittweise in neue Konzepte einzuführen. Und natürlich finden Sie die Beispiele auch auf dem beiliegenden Datenträger, um Ihnen die lästige Tipparbeit zu ersparen.

Die meisten Kapitel ranken sich um ein Hauptthema, scheuen sich aber auch nicht, interessante Nebenthemen aufzugreifen, wenn die Gelegenheit dafür günstig ist. Daher empfehlen wir Ihnen, daß Sie dieses Buch wie einen Kurs betrachten, der Sie innerhalb von rund zehn Tagen mit den Grundlagen der Programmiersprache Pascal vertraut machen kann.

Kapitel 1, Grundlagen

Dieses Kapitel beschäftigt sich mit den grundlegenden Sprachelementen eines Pascal-Programms und wie diese dargestellt werden. Daraus entstehen dann die ersten Programme, bei denen Sie auch schon einiges über Konstanten, Variablen und Datentypen kennenlernen.

Außerdem widmen wir uns der Ein- und Ausgabe von Daten, denn was nützen die schönsten Beispielprogramme, wenn man nicht mit ihnen »kommunizieren« kann. Da die Ein- und Ausgabefunktionen von Pascal recht simpel sind, brauchen wir danach nicht mehr weiter auf dieses Thema einzugehen.

Kapitel 2, Weltwissen

Dieses Kapitel befaßt sich mit Datentypen, Konstanten und Variablen, kurzum alles, was ein Programm benötigt, um sein Wissen über die Außenwelt verwalten zu können.

Gleichzeitig befassen wir uns hier auch mit den Möglichkeiten, Daten von der Tastatur aus einzugeben.

Kapitel 3, Mit Pascal können Sie rechnen

Daten werden in Programmen miteinander verknüpft. Dazu dienen Operatoren, die das Thema dieses Kapitels sind. Und damit kann man weit mehr machen als nur rechnen!

Kapitel 4, Programmsteuerung

Immer nur sequentielle Programme zu schreiben, ist langweilig. Spannend wird die Sache, wenn das Programm durch Verzweigungen und Schleifen Intelligenz erhält. Das eben wird in diesem Kapitel behandelt.

Kapitel 5, Öfter mal was Neues

Neben den vordefinierten Datentypen besteht für Programmierer auch die Möglichkeit, eigene Datentypen zu definieren. Das geht von einfachen Aufzählungen bis zu komplexen Datenstrukturen. Erst mit diesen Techniken lassen sich wirklich große Lösungen erstellen.

Kapitel 6, Dateizugriffe

Um Daten dauerhaft zu speichern, müssen Sie wissen, wie diese Daten auf der Festplatte in Dateien abgelegt und von dort wieder gelesen werden können. Pascal bietet für diese Aufgaben einige vordefinierte Strukturen und Funktionen an.

Und auch für den direkten Zugriff auf MS-DOS und seine Dateistruktur wird etwas geboten.

Kapitel 7, Teile und Herrsche

Komplexe Programme lassen sich dauerhaft nur unter Kontrolle bringen, wenn diese Komplexität in kleinere und besser zu überschauende Einheiten zerlegt werden kann.

Diese Einheiten werden dann mit Funktionen und Prozeduren realisiert. Diese können ihrerseits in Bibliotheksdateien gesammelt werden, was irgendwann dazu führt, daß Sie über einen großen Fundus an vordefinierten Lösungen verfügen!

Kapitel 8, Dynamische Speicherverwaltung

Mit Zeigern und den Funktionen `New()` und `Delete()` besteht die Möglichkeit, optimal auf die Programmbedürfnisse zugeschnittene Speicherstrukturen zu schaffen und zu verwalten. Dieses Kapitel zeigt Ihnen die Grundlagen der Zeiger und der dynamischen Speicherverwaltung. Zusammen mit den benutzerdefinierten Datentypen haben Sie ein sehr mächtiges Werkzeug in den Händen, um Programme zu schreiben.

Anhang

Im Anhang finden Sie eine tabellarische Übersicht über alle Schlüsselworte von Pascal sowie die wichtigsten Compilerdirektiven. Mit diesen Informationen sollten Sie dann in den Handbüchern zu Ihrem Compiler weiter nachschlagen.

Grundlagen 1

1.1 Pascal *20*

1.2 Syntax von Pascal *23*

1.3 Kommentare und Compilerdirektiven *27*

1.4 Programme *29*

1.5 Programmausgaben *33*

1.6 Ausblicke *38*

Im ersten Kapitel stellen wir die Sprachelemente eines Pascal-Programms vor und zeigen, wie diese dargestellt werden. Dazu gehört auch, nach welchen Regeln eigene Namen von Programmobjekten anzulegen sind.

Im Anschluß daran zeigen wir mit einem ersten kleinen Beispiel, wie ein Pascal-Programm prinzipiell aufgebaut ist. Darauf aufbauend werden Sie in den folgenden Kapiteln weitere Pascalprogramme gestalten.

1.1 Pascal

Pascal wurde in den 70er Jahren von N. Wirth als *Lehrsprache* konzipiert. Es sollte vor allem ein Gegenentwurf zu Programmiersprachen wie BASIC oder FORTRAN sein, die das Erlernen der Programmierung und ihrer Gesetzmäßigkeiten nicht vernünftig unterstützen.

Mit dem Compiler »Turbo-Pascal«, der allerdings oftmals von den ISO-Vorgaben zu Pascal abweicht, gab es auf den PCs zum ersten Mal einen schnellen und effektiven Compiler in einer freundlichen Entwicklungsumgebung.

Er war der Ausgangspunkt einer Entwicklung, die heute bis zur Version 7.0 des Borland Pascal Compilers geführt hat. Die grundlegenden Sprachregeln wurden dabei beibehalten und weiterentwickelt .

1.1.1 Was ist ein Pascal-Programm?

Ein Pascal-Programm besteht mindestens aus einer Datei, welche die Endung *PAS* hat, das Programm selber beginnt mit dem Wort `program`. Die Codedateien selber sind reine ASCII-Dateien. Daraus erzeugt der Compiler ein ausführbares Programm, eine *EXE*-Datei.

Während die Regeln, nach denen die Worte der Sprache miteinander kombiniert werden dürfen, festgelegt sind, ist die optische Gestaltung der Datei weitgehend frei. Sie können im Grunde verfahren, wie Sie

wollen, wenngleich es zweckmäßig ist, bestimmte Konventionen einzuhalten, um die Programmtexte besser lesen zu können.

Bibliotheken

Neben dieser Programmdatei gibt es die Möglichkeit, daß auch noch weitere Dateien mit vorbereiteten *Funktionen* und *Prozeduren* zu dem Programm hinzukommen.

Diese Dateien können beispielsweise von Ihnen geschrieben worden sein, um bestimmte, immer wieder benötigte Funktionen und Prozeduren als *Bibliothek* zu sammeln. Oder es handelt sich um fertige Bibliotheken, die Sie von irgendwoher bekommen haben.

Und zu guter Letzt kann es sich bei diesen Bibliotheken natürlich auch um die von Borland zu Pascal mitgelieferte *Standardbibliothek* handeln. Diese Sammlung an vorgefertigten Lösungen ist bei Borland Pascal traditionell recht umfangreich.

Mit einer einzigen Anweisung kann dem Compiler mitgeteilt werden, welche dieser Bibliotheksdateien, *Units* genannt, zu ihrem Programm hinzukommen sollen.

Der Übersetzungsvorgang

Damit ein in Pascal geschriebenes Programm vom Rechner ausgeführt werden kann, muß es in eine für die Maschine nutzbare Form gebracht werden.

Bei Borland-Pascal werden erst einmal alle Programmdateien auf ihre formale Richtigkeit überprüft – ein sogenannter *Syntaxcheck* – und dann in eine maschinenlesbare Form überführt. Danach wird eine spezielle Einheit – die *Laufzeitbibliothek* – dazugebunden und alle Einheiten werden mit dieser Laufzeitbibliothek zum fertigen Programm, einer *COM* oder *EXE*-Datei.

Diese Programmdatei wird von Ihnen oder einem anderen Benutzer des Programms aufgerufen, wenn das Programm gestartet werden soll. Meistens laufen die ersten Übersetzungsschritte und auch die späteren Tests nicht fehlerfrei ab, so daß Sie erneut die Programmtex-

te, die *Quelldateien*, in den Editor laden und bearbeiten müssen. Dann erfolgt wieder die Übersetzung in eine Programmdatei. Den Übersetzungsschritt nennt man die *Compilierung* des Programms.

1.1.2 Beschreibung der Programm-Elemente

Das Pascal-Programm selber ist eine reine Textdatei, in der Befehle und Ausdrücke nach bestimmten – durch die *Syntax* der Programmiersprache festgelegten – Regeln niedergeschrieben sind.

Wenn es sich um kompliziertere Konstrukte handelt, kann deren textuelle Beschreibung schnell schon einmal ein Wust von Worten und Sätzen werden, der eher schlecht zu lesen ist. Daher gibt es die sogenannten *Syntaxdiagramme*, die wir Ihnen hier vorstellen wollen.

Bitte verwechseln Sie Syntaxdiagramme nicht mit Struktogrammen, welche die Ablauflogik eines Programms nachbilden.

Regeln der Syntaxdiagramme

Ein Syntaxdiagramm dient dazu, den Aufbau eines Befehls und den Kontext, in dem er eingesetzt werden kann, auf einfache Art zu beschreiben, ohne in allzuviele Worte verfallen zu müssen. Dabei besteht ein Syntaxdiagramm aus drei Grundkomponenten:

- **Linie/Pfeil**: Zeigt den Zusammenhang der einzelnen Bauteile eines Sprachkonstrukts auf. Man folgt den Linien eines Syntaxdiagramms von links nach rechts, dabei Verzweigungen durchlaufend.
- **Kästchen**: In einem Rechteck stehen die zu beschreibenden Konstrukte. Dabei sind Syntaxdiagramme *rekursiv* aufgebaut, daß heißt, in einer Konstruktbeschreibung können sehr wohl weitere Konstrukte, die irgendwo anders eine eigene Beschreibung stehen haben, benutzt werden. Wenn man die Bedeutung dieses verwendeten Konstrukts wissen will, muß man eben dessen Syntaxdiagramm suchen.

- **Kreise/Ovale**: In ihnen stehen *Schlüsselworte, Operatoren, Interpunktionszeichen* und auf der untersten Ebene der Syntaxdiagramme Zeichen des *Alphabets.*

So wie es in der Pascal-Hilfe üblich ist, werden auch wir im Rahmen des Buches von Fall zu Fall Syntaxdiagramme zur Beschreibung von Sprachkonstrukten benutzen.

Sprache in Syntaxdiagrammen: Wir werden als Sprache der Syntaxdiagramme Englisch verwenden. Der tiefere Grund dafür ist, daß Pascal ebenfalls englische Worte benutzt.

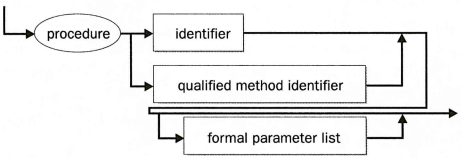


Bild 1.1: Beispiel für ein Syntaxdiagramm

1.2 Syntax von Pascal

Die einfachsten Grundelemente einer jeden Programmiersprache sind die erlaubten Zeichen *(Alphabet)*, die vordefinierten Worte, sowie die damit zusammenhängenden Namensregeln.

1.2.1 Im Programm erlaubte Zeichen

Das Grundalphabet der Programmiersprache Pascal sind 7 Bit ASCII-Zeichen. Das bedeutet, daß Sonderzeichen, wie Umlaute oder grafische Zeichen beispielsweise, innerhalb von Funktionsnamen nicht zulässig sind:

```
0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M N O P Q R S T U  V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
< > = . ^ @ * + - / , : ; { } ( ) [ ] # $ & _ '
```

Zu obiger Liste kommen noch die sogenannten *Whitespaces* hinzu. Das sind Leerzeichen, der Tabulatorschritt und der Zeilenumbruch. Die Einschränkung, daß nur die oben angegebenen Zeichen benutzt werden dürfen, gilt natürlich nur für Konstruktionen, die innerhalb der Programmiersprache eingesetzt werden.

In Zeichenketten erlaubte Zeichen

Für *Zeichenketten,* die einen auszugebenden Text enthalten, können Sie alle beliebigen, darstellbaren Zeichen eintragen. Die Zeichenkette wird vom Compiler nämlich nicht beachtet und als ein Ganzes ohne Berücksichtigung seines Inhaltes benutzt. Eine solche Zeichenkette wird zwischen zwei Hochkommas gesetzt. Ein anderer Name für Zeichenkette ist *String.* Nachfolgend ein paar Zeichenketten, die nach den Regeln von Pascal vollkommen korrekt sind:

```
'Dies ist eine Zeichenkette'
'0 1 2 3 4 5 6 7 8 9'
'Umlaute (äöü) dürfen nur in Zeichenketten vorkommen'
'---> Ausgabetext <---'
```

1.2.2 Schlüsselworte und Standardanweisungen

Pascal kennt eine große Zahl von *Schlüsselworten.* Das Besondere bei diesen reservierten Schlüsselworten ist, daß Sie sie innerhalb von Programmen nicht für Ihre eigenen Programmobjekte benutzen dürfen. Es ist also nicht zulässig, beispielsweise das Schlüsselwort `while` als Variablennamen benutzen zu wollen. Das gilt, da Pascal nicht zwischen der Groß- und Kleinschreibung unterscheidet, natürlich auch dann, wenn Sie es verfremdet mit Großbuchstaben schreiben. Im Anhang finden Sie eine Tabelle mit den Schlüsselwörtern von Borland-Pascal 7.0. Wir haben versucht, sie in Kategorien einzuordnen, die mit ihrer Benutzung verbunden sind.

Wundern Sie sich nicht, wenn auch einige Ihnen recht fremde Worte auftauchen, die wir in diesem Buch nicht nutzen. Es handelt sich dabei um Schlüsselworte zur objektorientierten Programmierung, die mit Borland Pascal ebenfalls möglich ist.

Standardanweisungen

Neben den reservierten Worten gibt es auch noch die *Standardanweisungen*. Dabei handelt es sich um eine Gruppe von Worten, die zwar vorgegeben sind, jedoch bei Bedarf mit einer neuen Bedeutung versehen werden können. Ob das allerdings eine gute Idee ist, darf einigermaßen bezweifelt werden, weswegen wir diese Standardanweisungen mit dem gleichen Respekt benutzen werden wie die reservierten Worte. Der Anhang A enthält eine Tabelle mit den bekannten Standardanweisungen.

1.2.3 Namensregeln

Um Variablen, Funktionen oder Prozeduren zu bezeichnen, vergibt man Namen. Die Gestaltung dieser Bezeichner unterliegt festen Regeln.

Regeln für Bezeichner

Natürlich beziehen sich diese Regeln nur auf Namen, die im Programm genutzt werden, nicht jedoch auf das, was Sie innerhalb von Zeichenketten eintragen.

- Ein Name beginnt mit einem Buchstaben oder einem Unterstrich.
- Im Namen sind Ziffern, Buchstaben und der Unterstrich erlaubt.
- Namen können beliebig lang sein, es werden aber nur bis zu 62 Zeichen unterschieden.

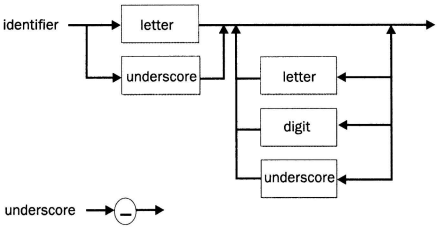


Bild 1.2: Namensregeln als Syntaxdiagramm

Namenskonventionen

Die kürzesten Bezeichner bestehen somit aus einem einzigen Buchstaben, sind dann aber auch nicht übermäßig aussagekräftig. Das andere Extrem wären sehr lange Namen. Solche Namen würden ein Programm auch nicht leserlicher machen, zumal die wirkliche Unterscheidungsgrenze ja bei 62 Zeichen liegt.

Sie müssen also im Rahmen der gegebenen Möglichkeiten einen Kompromiß schließen zwischen guter Tippbarkeit und guter Lesbarkeit. So hat es sich in Pascal eingebürgert, daß man kurze, aus Verben und Hauptwörtern bestehende Namen benutzt. Wenn es sinnvoll ist, werden diese miteinander kombiniert. Dabei kann man verschiedenen Konventionen folgen, beispielsweise alles klein schreiben, was gerade bei längeren Namen recht unangenehm lesbar ist:

```
diesefunktionmachtwas()
```

Oder mit Unterstrichen, um einzelne Worte zu trennen (Leerzeichen sind ja in Namen nicht erlaubt!).

```
diese_funktion_macht_was()
```

Eine weitere Möglichkeit wäre es, Wortanfänge innerhalb der Namen mit einem Großbuchstaben zu versehen. Wobei es Pascal allerdings egal ist, wie Sie den Namen schreiben, da keine Unterscheidung zwischen der Groß- und Kleinschreibung erfolgt.

```
DieseFunktionMachtWas()
```

Egal welche Konvention Sie befolgen, wichtig ist, daß der Name einen gut lesbaren Rückschluß auf das damit zusammenhängende Programmobjekt zuläßt.

Konventionen in diesem Buch

In diesem Buch werden wir es so halten, daß alle originalen Pascalworte klein geschrieben werden. Bei den Funktionen und Prozeduren, sowie den vordefinierten Konstanten halten wir uns an die von Borland in der Hilfe und der Dokumentation gegebenen Schreibweisen.

Eigene Funktionen, Prozeduren und Variablen werden wir im Stile von `DatenQuelle` schreiben. Also als zusammenhängende Worte, wobei jedes Wort, auch wenn es ein Verb oder ähnliches ist, mit einem Großbuchstaben beginnt (`IstLeer`, `BerechnungsProzedur`). Einbuchstabige Namen werden wir nur für Hilfs- und Zählvariablen benutzen. Alle Objekte, die Daten für einen längeren Zeitraum und unterschiedliche Bearbeitungsschritte enthalten sollen, werden mit sinnvollen Namen belegt werden.

Die Schreibweise von Konstanten möchten wir immer in Großbuchstaben vornehmen, um sie im Programmtext besser von anderen Programmobjekten unterscheiden zu können.

Welche Konventionen Sie später für Ihre Programme nutzen, sei Ihnen überlassen, Hauptsache ist, Sie entscheiden sich für einen durchgängigen Stil und achten auf gute Lesbarkeit.

1.3 Kommentare und Compilerdirektiven

Bei größeren Programmen mit komplexen Befehlsfolgen ist es eine gute Idee, an entscheidenden Stellen *Kommentare* in das Programm einzufügen. Damit läßt sich die Verständlichkeit für den Leser des Codes erhöhen.

In Pascal sind zwei Arten von Kommentaren definiert, die allerdings aus historischen Gründen auch noch für die Steuerung des Compilers benutzt werden können.

Der Kommentarblock

Die Kommentare funktionieren in Pascal immer als Kommentarblock. Das heißt, daß nach einem Kommentarbeginn alles bis zum Kommentarende ausgeblendet wird, wobei der Kommentar auch mehrzeilig sein darf.

```
{ ... }
(* ... *)
```

Beachten Sie bitte, daß zwischen der runden Klammer und dem Stern keine Leerzeichen erlaubt sind! Schachtelungen der Kommentare sind ebenfalls erlaubt, was es möglich macht, Kommentare ganz gut zu Zwecken der Fehlersuche einzusetzen. Es ist nicht erlaubt, die Kommentare wechselseitig gemischt einzusetzen:

Erlaubt:

```
{ ... }
{ (* ... *) }
{ { ... } }
(* (* ... *) *)
(* { ... } *)
```

Verboten:

```
(* ... }
{ ... *)
```

Compilerdirektiven

Neben diesen Grundelementen findet man in Pascal-Programmen auch noch die schon erwähnten *Compilerdirektiven,* die direkt im Programmtext angeben, wie sich der Compiler ab dieser Direktive im Rahmen der Übersetzung verhalten soll, welche Art von Code er erzeugen soll usw.

Eine Compilerdirektive steht immer innerhalb von Kommentarzeichen (`(*...*)` oder `{ }`) und beginnt mit einem $-Zeichen. Auf die Liste der möglichen Compilerdirektiven möchten wir hier nicht eingehen und verweisen Sie deshalb auf die Hilfe zu Borland-Pascal, Stichwort Compilerbefehle. In den meisten Fällen werden Sie sie wohl noch nicht einsetzen und sich eher auf die in der Compilerumgebung gesetzten Parameter verlassen. Aber Profis können mit den individuell zu setzenden Compilerdirektiven etliche interessante Dinge machen!

1.4 Programme

Genug der Definitionen und Worte, lassen Sie uns unsere ersten Pascal-Programme schreiben. Sie werden sehr kurz sein und nicht allzu viel nützliche Dinge machen, aber es sind vollständige Programme nach den Regeln der Sprache.

Und Sie lernen mit ihnen schon soviel über Pascal kennen, daß Sie erste eigene Experimente machen können. Und das sollen Sie auch, denn gerade durch die Fehler, die Ihnen dabei unterlaufen, werden Sie Pascal recht schnell »in den Griff bekommen«.

1.4.1 Der Programmaufbau

Das erste Programm ist ein absolut leeres Programm. Es macht wirklich gar nichts, außer daß es sich klaglos übersetzen läßt. Die entstehende Programmdatei können Sie auch gerne starten, nur wie gesagt, passieren wird nichts.

Ein minimales Programm

Wir zeigen Ihnen das Programm, weil es der Rahmen ist, in den später alle folgenden Beispiele eingefügt werden. Für alle Pascal-Programme wird der gleiche Rahmen benutzt werden. Und so sieht es aus, das allereinfachste Pascal-Programm:

```
1 program Name (input, output);
2
3 begin
4
5 end.
```

Listing 1.1: Das leere Programm (💾 1_1.PAS)

Numerierung der Programmzeilen: In Pascal ist es weder nötig, noch erlaubt, die einzelnen Programmzeilen mit einer Nummer zu versehen. Wir werden diese Numerierung vornehmen, um uns in den dazugehörigen Texten besser auf einzelne Programmteile beziehen zu können. Wenn Sie diese Beispiele abtippen, dürfen Sie die Nummern **nicht** übernehmen, sonst gibt es Compilerfehler!

Die erste Zeile des Programms enthält das reservierte Wort `program`, gefolgt von einem beliebigen Programmnamen, der übrigens nicht mit dem Namen der Programmdatei übereinstimmen muß. Danach kommt in runden Klammern eine Information, daß das Programm die *Ein- und Ausgabekanäle* benutzen will.

Nach diesem Programmbeginn werden in nutzvolleren Beispielen als diesem weitere Informationen zum Programm stehen, wie Variablenvereinbarungen, Typdefinitionen usw. Jetzt ist hier nur die leere Zeile 2 zu finden.

Der Hauptteil des Programms, gewissermaßen der *Programmrumpf,* besteht aus den Worten `begin` und `end` in den Zeilen 3 und 5, zwischen denen sich später alles »Leben« der Programme abspielen wird. Beachten Sie bitte, daß wir die erste Zeile mit einem Semikolon (;) beenden, und daß hinter dem letzten `end` immer ein Punkt steht. Hinter dem `begin` darf dagegen kein Semikolon stehen, da es mit dem `end` zusammen eine Einheit bildet.

Die Borland Version

Obiges Programm entspricht sogar noch den Regeln für klassisches Pascal. Nach den Regeln von Borland kann es noch kürzer sein und besteht dann nur noch aus den folgenden Zeilen.

```
1 begin
2     ...
3 end.
```

Listing 1.2: Das kürzeste denkbare Programm

Wir werden unsere Programme alle nach einer Mischform aus klassischem Stil und dem Borland-Stil aufbauen, der auch von den meisten anderen Programmierern befolgt wird:

```
1 program Name;
2
3 begin
4     ...
5 end.
```

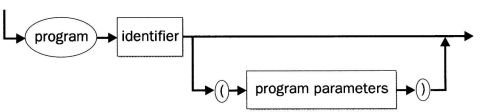


Bild 1.3: Syntaxdiagramm zu program

Wie Sie sich sicherlich denken können, gehören zu einem brauchbaren Programm noch sehr viel mehr Bausteine. In den folgenden Kapiteln werden Sie diese alle kennenlernen.

1.4.2 Einsatz externer Bibliotheken

Eine erste Erweiterung des Programmrumpfes erfolgt, indem der Zugriff auf andere zu dem Programm gehörige Dateien ermöglicht wird. Wir haben ja schon darauf hingewiesen, daß ein fertiges Programm sich aus mehreren Dateien aufbauen kann, die jede für sich getrennt angelegt und bearbeitet werden.

Die uses-Anweisung

Wenn Sie Teile dieser Dateien benutzen wollen, müssen Sie dem Compiler mitteilen, welche Einheit Sie denn meinen. Und das machen Sie mit einer `uses`-Anweisung, der eine durch Kommas getrennte Liste der Units folgt. Abgeschlossen wird diese Liste durch ein Semikolon.

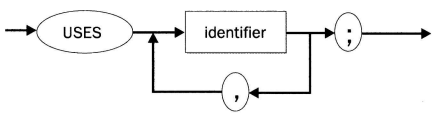


Bild 1.4: Syntaxdiagramm für die uses-Anweisung

```
program Name;

uses
    ...;

begin
    ...
end.
```

Die Unit WinCrt

Ein Grund, sich jetzt schon mit Units zu beschäftigen, liegt darin, daß Sie vielleicht die Windows-Version der IDE benutzen. Und Windows hat seine eigenen Ausgabeanweisungen und -verfahren, die

mit den einfachen, unter DOS zu nutzenden Kommandos nicht mehr zusammenpassen.

Wir werden alle unsere Programmbeispiele in der DOS-Version darstellen. Sollten Sie diese 1:1 übernehmen, müssen Sie die unter DOS üblichen Ein- und Ausgabefunktionen auf Windows abbilden. Damit das nicht zu aufwendig wird, hat Borland eine Unit namens `WinCrt` entwickelt, die in einem Fenster das Verhalten des Textbildschirms unter DOS simuliert. Damit werden Sie zwar keine Windows-Programme schreiben, können aber alle DOS-Ein- und -Ausgabeanweisungen auch in diesem Fenster nutzen und somit allen unseren Programmbeispielen ohne Schwierigkeiten folgen.

```
1 program Name;
2
3 uses
4     WinCrt;
5
6 begin
7     WriteLn('Hallöchen');
8 end.
```

Listing 1.3: Ein windowsfähiges Programm (🖫 1_2.PAS)

1.5 Programmausgaben

Im obigen Beispiel haben Sie schon das erste Pascal-Programm mit einer Ausgabe kennengelernt. Es macht eine einfache Textausgabe. Dazu bedient es sich der Ausgabefunktion `WriteLn()`.

Da dieses Thema für alles weitere recht wichtig ist, widmen sich die folgenden Beispiele den verschiedenen Aspekten der Ausgabe von Programminformationen. Um die Eingabe kümmern wir uns im nächsten Kapitel, wenn wir die ersten Variablen einführen.

1.5.1 Nette Grüße

Einfache Textzeilen können Sie mit den Funktionen `Write()` und `WriteLn()` ausgeben. Das Ausgabeziel ist die sogenannte Standardausgabe, die ihrerseits mit dem Bildschirm verbunden ist. Die beiden Funktionen können auch dazu genutzt werden, um in eine Datei zu schreiben, was wir allerdings erst später ausnutzen wollen.

Ausgabe mit WriteLn()

Das folgende Programm benutzt daher `WriteLn()` für einen freundlichen Gruß an die Welt:

```
1 program Hello;
2
3 begin
4     WriteLn('Hello, world');
5 end.
```

Listing 1.4: Ausgabe einer Zeichenkette (1_3.PAS)

Die Zeichenkette »`Hello, world`« wird als *Stringkonstante* bezeichnet, da ihr Inhalt in diesem Programm fest verdrahtet ist. Eine Stringkonstante wird immer in zwei einfache Anführungszeichen eingefügt. Mit `WriteLn()` können Sie neben Texten auch andere Werte formatiert ausgeben.

Der Unterschied zwischen WriteLn() und Write()

Die Funktion `WriteLn()` schickt nach der Ausgabe automatisch einen Zeilenumbruch nach, weswegen sie auch die Namensendung `ln` für *Linefeed-Newline* trägt. Sollten Sie `WriteLn()` ohne Argumente benutzen, lassen Sie die Klammern weg und als Ergebnis erhalten Sie einen einfachen Zeilenumbruch.

Eng mit `WriteLn()` verwandt ist `Write()`. Hier erfolgt die Ausgabe, ohne daß ein Zeilenumbruch gesetzt wird. Eine Folgeausgabe geht also direkt in der gleichen Zeile weiter.

```
1 program HelloNummerZwei;
2
3 begin
4     Write('Hello, ');
5     Write('world');
6     WriteLn;
7 end.
```

***Listing 1.5: Einsatz von Write()* (1_4.PAS)**

Beachten Sie bitte, daß wir die einzelnen Ausgabezeilen immer mit einem Semikolon abgeschlossen haben. In Pascal gilt nämlich die Regel, daß eine Anweisung durch ein Semikolon abgeschlossen werden muß. Eine Ausnahme ist der Fall, bei dem sich aus dem Zusammenhang, in dem das Konstrukt steht, von selber ergibt, daß die Anweisung abgeschlossen ist.

Das ist bei den obigen Beispielen immer bei der letzten Anweisungszeile vor dem `end` der Fall. Ein Semikolon schadet hier jedoch nichts und daher haben wir es gesetzt.

Gemischte Text- und Zahlenausgabe

Zahlenausgaben können bei `WriteLn()` und `Write()` gemischt mit Text oder ganz für sich alleine erfolgen. Überhaupt ist es möglich, verschiedene Programmelemente, wie die im nächsten Kapitel beschriebenen Konstanten und Variablen, in einer Ausgabeanweisung beliebig miteinander zu verbinden.

Dazu müssen Sie die einzelnen Komponenten lediglich durch Kommata voneinander trennen. Es ist sogar erlaubt, *Berechnungsausdrücke* oder *Funktionsaufrufe* innerhalb der runden Klammern von `Write()` und `WriteLn()` einzusetzen. Die Ergebnisse der Berechnung oder das, was die Funktion zurückliefert, wird dann in einer passenden Form ausgegeben.

```
Write('Text', 5, variable, FunktionsAufruf, 1 + 1);
```

Im folgenden Beispiel mischen wir konstante Texte, eine Berechnung und konstante Zahlen miteinander und lassen alles ausgeben.

```
1 program Mischmasch;
2
3 begin
4     WriteLn('Jetzt kommen ein paar Zahlen :');
5     WriteLn('Erste Zahl:', 5);
6     WriteLn('Zweite Zahl: ', 6.66);
7     WriteLn('Deren Summe ist: ', 5 + 6.66);
8 end.
```

Listing 1.6: Berechnete Ausdrücke in der Ausgabe (🖫 **1_5.PAS**)

1.5.2 Formatierte Ausgabe

Wie Sie bei der Ausführung des Beispiels sehen können, werden die Zahlen in diesem Fall als sogenannte *numerische Konstanten* angegeben und durch Kommata von den Zeichenfolgen getrennt, die ebenfalls ausgegeben werden. Die erste Zahl ist eine ganzzahlige Konstante und die zweite Zahl eine *Fließkommakonstante;* mit dem Punkt als Dezimaltrennzeichen. Das Berechnungsergebnis ist ebenfalls eine Fließkommazahl, da die einfache Ganzzahl vorher intern angepaßt wird.

Automatische Formatierung von Fließkommawerten

Die Fließkommawerte werden in einem Standardformat ausgegeben, welches als *Exponentialformat* bezeichnet wird:

```
6.6600000000E+00.
```

Die Anzahl der Nachkommastellen ist hier eher unsinnig, gleiches gilt für die Darstellung der Zahl als Vielfaches zur Basis 10. `E` steht hier für den Zehnerexponenten, nicht für die Eulersche Zahl. 10^0 entspricht dann der Multiplikation mit 1.

Wenn Sie bei der Ausgabe von Fließkommazahlen eine Formatierungsinformation angeben, können Sie dieses unschöne Erscheinungsbild vermeiden:

```
1 program Formatierung;
2
3 begin
4     Write('Jetzt kommt eine formatierte Zahl: ');
5     WriteLn(6.66:2);
6     Write('Und noch einmal: ');
7     WriteLn(6.66:2:2);
8 end.
```

Listing 1.7: Formatierte Zahlenausgabe (1_6.PAS)

In diesem Fall werden der Text und die Zahl in einer Zeile ausgegeben, weil wir `Write()` als Ausgabefunktion für die Textzeile nutzen. Die Zahl wird formatiert mit `WriteLn()` ausgegeben und die nächste Ausgabe kommt so wieder in eine eigene Zeile.

Formatierungsmöglichkeiten: Gesamtplatz

Die Formatierung bestimmt, wieviel Platz der Zahl zur Verfügung gestellt wird und wie viele *Nachkommastellen* dargestellt werden sollen. Handelt es sich um eine Fließkommazahl und geben Sie keine Nachkommastellen an, wird zur Not gerundet sowie die Exponentialschreibweise gewählt. Ist der Darstellungsplatz größer als die Zahl, wird sie rechtsbündig angeordnet:

```
6.66:2          '6.7E+00'
6.66:10         ' 6.66E+00'
6.66:20         '    6.6600000000E+00'
6.66:30         '                6.6600000000E+00'
```

Die erste Formatierung gibt also an, wie viele Stellen die Zahlendarstellung haben darf. Diese Formatierung, mit der eine rechtsbündige Ausrichtung zu erreichen ist, dürfen Sie übrigens auch für Zeichenketten benutzen.

Formatierungsmöglichkeiten: Dezimalstellen

Neben dem Ausgabeplatz können Sie bei numerischen Werten auch die Nachkommastellen angeben. Und zwar indem Sie zuerst die Darstellungsbreite und dann die Nachkommastellen durch Doppelpunkte getrennt an die Zahl hängen:

```
6.66:2:2      '6.66'
6.66:10:2     '      6.66'
6.66:20:2     '                 6.66'
6.66:30:2     '                           6.66'
```

1.6 Ausblicke

Nach diesem ersten Kapitel kennen Sie schon die Grundregeln einfacher Pascal-Programme. Erste Ausgaben sind auch schon erfolgreich möglich gewesen.

Was allerdings fehlt, um brauchbare Programme zu schreiben, sind Variablen, mit denen Informationen zwischengespeichert werden können und Kontrollstrukturen, mit denen der Programmablauf sich an unterschiedliche Gegebenheiten anpassen kann.

Dies wird dann auch das Thema des nächsten Kapitels sein.

Weltwissen 2

2.1 Was sind Datentypen? 40

2.2 Konstante und Variablen 46

2.3 Die Eigenschaften ordinaler Datentypen 54

2.4 Reale Datentypen 63

2.5 Zeichenketten 64

2.6 Kleines Intermezzo: Dateneingabe 68

Das vorige Kapitel brachte Ihnen die ersten Elemente der Programmierung in Pascal näher. Allerdings sind Programme, die immer nur feste Werte ausgeben, sicherlich kein besonderer Fortschritt.

Daher soll Ihnen dieses Kapitel zeigen, wie Sie ihre Programme mit den in Pascal eingebauten Möglichkeiten Daten zu verwalten, erweitern können.

Es geht dabei um *Datentypen, Konstanten, Variablen.* In dem Zusammenhang bietet es sich dann auch an, die Verfahren zur Dateneingabe von der Tastatur anzusprechen. Sie lernen nun also die wichtigsten grundlegenden Datentypen und ihre Verwendung kennen.

Das nächste Kapitel wird sich dann den *Operatoren* zuwenden, mit denen all diese Daten zu neuen, sinnvollen Ergebnissen verknüpft werden können.

Sollte Ihnen dieses Kapitel zu trocken sein, picken Sie sich am besten nur das heraus, was Sie jetzt interessiert und kehren später ganz nach Bedarf noch einmal hierher zurück.

2.1 Was sind Datentypen?

Alle Programme in diesem Buch – außer denen des vorherigen Kapitels – lesen Daten, speichern und verändern sie, und können die Ergebnisse ihrer Berechnungen wieder ausgeben. Im Grunde funktionieren daher alle Programme nach dem gleichen Prinzip:

- Eingabe
- Verarbeitung
- Ausgabe

Um *EVA* zu realisieren, bedarf es Möglichkeiten, mit denen Daten vom Benutzer oder anderen Datenquellen entgegengenommen werden können, um im Programm selber vorrätig gehalten zu werden. Und so wird auch Platz benötigt, um Zwischenberechnungen auszuführen.

Damit kommen wir also auf die *variablen Speicherplätze*, kurz *Variablen* genannt, zu sprechen. Sie sind gewissermaßen die Behälter, in denen die Information während des Programmlaufs liegt und wo sie entnommen werden kann.

Wo variable Werte existieren, gibt es auch konstante Werte. Dabei kann es sich um Naturkonstanten handeln, oder auch um menschgemachte Werte, wie die (derzeit noch) konstante Mehrwertsteuer. Die Darstellung konstanter Werte ist also auch ein Thema für die Programmierung.

All diese Konstrukte basieren aber auf einem noch grundlegenderen Modell, nämlich dem des Datentyps.

2.1.1 Die Idee der Datentypen

Um Daten im Programm korrekt verwalten und einsetzen zu können, braucht jede Programmiersprache eine Art *Vorstellung von Datentypen*. So ist es offensichtlich, daß einzelne Zeichen und Zeichenketten eine andere Art von Daten sind, als beispielsweise Ganz- oder Fließkommazahlen. Gar nicht zu reden von Datentypen, die der Programmierer sich definiert hat.

Datentypen geben dem Compiler somit Informationen über die folgenden Dinge:

- Benötigter Speicherplatz
- Erlaubtes Speicherformat
- Erlaubte Operationen mit den Daten

Die Konsequenz daraus ist die, daß bei der Übersetzung sowie zur Laufzeit eines Programms Fehler abgefangen werden können, die durch die falsche Verwendung von Datentypen in Ausdrücken wie »`'Willi'` + `99` entstehen.

2.1.2 Vordefinierte Datentypen

Die in Pascal vordefinierten *Standarddatentypen* sind für *Ganzzahlen*, *Fließkommazahlen* und *Zeichenfolgen* gedacht. Im Rahmen der im PC-Bereich möglichen verschiedenen Größen für Ganz- und Fließkommazahlen, sowie der Genauigkeit der Fließkommazahlen, existieren von diesen Datentypen mehrere Varianten.

Hier folgt erst einmal die Liste der vordefinierten Urdatentypen und ihrer Namen:

Tabelle 2.1: Ursprüngliche Datentypen in Pascal

Name	Bedeutung
char	Ein einzelnes Zeichen
string	Zeichenkette, bis zu 255 Zeichen am Stück
integer	Ganzzahl, Wertebereich von -32768 bis 32767
real	Fließkommazahl, Wertebereich von $-1{,}7\ 10^{39}$ bis $1{,}7\ 10^{38}$
boolean	Wahrheitswerte, deren Werte true oder false lauten

Ordinale Datentypen

Neben der Unterscheidung nach Art der Daten gibt es noch eine Unterscheidung nach der Art der Datenspeicherung. Es handelt sich um den Unterschied zwischen *ordinalen* und *realen* Datentypen.

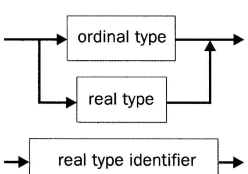


Bild 2.1: Einfache Typen im Syntaxdiagramm

Die Unterscheidung zwischen ordinalen und realen Datentypen bezieht sich darauf, ob eine Aussage wie die folgende möglich ist:

»x ist der Nachfolger von y«

So ist 1 mit Sicherheit der Nachfolgewert von 0. Und eine Position weiter landen Sie bei der 2. Da Zeichen und generell alle Symbole bei Computern ebenfalls durch eine Abbildung auf die Zahlen 0 bis 255 verwaltet werden, kann man auch sagen, daß »a nach b« und »D nach C« kommen. Es gibt also auch bei den Zeichen für jedes Symbol einen eindeutigen Nachfolger. Jedes Symbol liegt in einer *genauen und eindeutigen Anordnung* zu seinen Mitsymbolen vor. Es hat eine eindeutige Position mit einem genau bestimmbaren Vorgänger und Nachfolger.

Solche genau unterscheidbaren Wertemengen werden in Pascal *Ordinalzahlen* genannt. Ganzzahlen sind somit ordinale Datentypen, genauso wie Zeichen und auch die logischen *(boolschen)* Werte `True` und `False`.

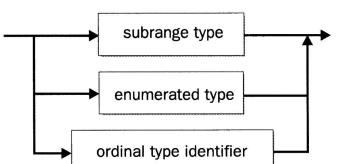


Bild 2.2: Syntaxdiagramm ordinaler Typen

Reale Datentypen

Etwas anders sieht die Sache aus, wenn Sie Fließkommazahlen betrachten. Denn hier ist die Frage nach dem Vorgänger oder Nachfolger einer Zahl nicht zu beantworten. Wer ist der Vorgänger von 1.000? Etwa 0.0000, oder 0.0001, 0.9999, 0.99999, 0.999999 ... ?

Das bedeutet, daß die Menge der darstellbaren *Werte nicht mehr eindeutig abzählbar* ist. Je nach gewünschter Genauigkeit läßt sich zwi-

schen zwei einzelnen Werten noch eine unendliche Folge weiterer Werte finden, die zwischen sie passen. Es gibt keine exakte Positionierung mehr, keine eindeutigen Vorgänger oder Nachfolger. Genau das sind in Pascal die realen Datentypen.

In der Urversion von Pascal gab es deshalb auch nur den Datentyp `real`, der allerdings durch Borland aus praktischen Gründen etwas ergänzt wurde. Die Ursache für diese Ergänzung lag vor allem in der beschränkten Genauigkeit, die der Rechner für Realzahlen zur Verfügung stellt.

2.1.3 Besondere Datentypen

Neben den einfachen ordinalen und realen Datentypen kennt Pascal noch einige weitere vordefinierte Datentypen, die sich nicht so leicht einordnen lassen.

Zeichenketten

Da sind beispielsweise die *Zeichenketten* zu nennen. Es handelt sich dabei um Anordnungen von Zeichen aus dem *ASCII-Alphabet,* die einen Text ergeben. Die Art, wie diese Texte gespeichert werden, kann nach zwei unterschiedlichen Verfahren erfolgen.

Das klassische Verfahren bietet jeder Zeichenkette maximal 255 Zeichen Platz an. Die aktuelle Länge der Zeichenkette wird innerhalb des Speicherplatzes auf der ersten Byteposition vermerkt.

Daneben gibt es eine Erweiterung von Borland, die sich an der Art orientiert, wie C/C++ Zeichenketten organisieren. Hier können die Texte im Prinzip beliebig groß werden.

Dateitypen

Dann gibt es noch die *Dateitypen*, mit denen die Voraussetzungen geschaffen werden, damit sich ein Programm an eine Datei anhängen und sie lesen oder schreiben kann. Dieses Lesen oder Schreiben erfolgt in Kombination mit den anderen Datentypen, egal ob es sich dabei um vordefinierte oder benutzerdefinierte Datentypen handelt.

Zeiger

Eine ganz besondere Klasse von Datentyp ist der *Zeiger (Pointer)*. Dabei handelt es sich im Grunde um nichts weiter als eine Speicheradresse im RAM.

Über einen Zeiger wird es möglich, auf die Inhalte zuzugreifen, die ab dieser Speicherstelle vorliegen. Je nach Art und Weise, wie der Zeiger definiert wurde, kann das ein typisierter oder untypisierter Zugriff sein.

Benutzt werden solche Zeiger immer dann, wenn eine *dynamische Speicherverwaltung* nötig ist. Beispiele dafür sind verkettete Listen, dynamische Arrays oder auch Datenbäume.

Im letzten Kapitel des Buchs werden Sie sowohl mit den Zeigern als auch den Möglichkeiten der dynamischen Speicherverwaltung Bekanntschaft schließen können.

2.1.4 Benutzerdefinierte Datentypen

Da es für leistungsfähige Programme wichtig ist, die Daten möglichst effektiv zu handhaben, bietet Pascal die Möglichkeit an noch weitere Datentypen einzusetzen.

Benutzerdefinierte Datentypen

Diese Datentypen werden je nach Bedarf durch einen Programmierer aufgebaut und können dann in weiten Bereichen genauso eingesetzt werden, als wenn es sie schon immer in der Form gegeben hätte.

Dazu gehören *Mengen (Set)*, *Felder (Array)* und vor allem auch komplexe Datenstrukturen *(Record, Variant)*.

Auch wenn mit selbstdefinierten Datentypen wiederum neue selbstdefinierte Datentypen angelegt werden können, basieren auch sie letztlich alle auf den vordefinierten Grunddatentypen.

Im weiteren Verlauf des Buchs werden Sie noch reichlich auf diese selbstdefinierten Datentypen treffen können.

Klassen und Objekte

Die objektorientierte Programmierung hat mit dem Konzept der *Klassen* noch eins draufgesetzt. Auch hier ist es das erklärte Ziel, daß der Programmierer komplexe Sachverhalte auf möglichst einfache und effektive Art aufrufen und ausdrücken kann.

Diese objektorientierten Erweiterungen von Pascal sind allerdings nicht unser Thema und bleiben einem anderen Buch vorbehalten.

2.2 Konstante und Variablen

Ein wichtiger Aspekt für die Entwicklung sinnvoller Programme sind die Unterschiede und die Nutzung von *Variablen* und *Konstanten.* Mit ihnen und den Operatoren für die Datentypen wird es möglich, sinnvolle Programme zu schreiben.

2.2.1 Literale Konstanten

Eine *Konstante* ist ein feststehender Wert. Dieser Wert, beispielsweise die Kreiszahl Pi (3.141...) wird in Berechnungen immer gleich benutzt und darf nicht geändert werden. Grundsätzlich werden zwei Arten unterschieden, um mit Konstanten umzugehen.

Das Konzept der literalen Konstante

Wenn Sie den Wert selber angeben, beispielsweise 3.141, nennt man diese für sich selber stehende Fließkommazahl eine *literale Konstante.*

Innerhalb des Programmcodes werden Sie sie in Berechnungen an den benötigten Stellen immer gleich einsetzen.

```
...
erg := r * 3.141;
...
erg := rad / 3.141;
...
```

Schreibweisen von Ganzzahlkonstanten

Die Darstellung kann wahlweise im bekannten Dezimalsystem vorgenommen werden oder als *Hexadezimalzahl*. Die Darstellung im Dezimalsystem folgt den gängigen Konventionen:

Bei der Darstellung von Hexadezimalzahlen müssen Sie der Zahl ein `$`-Zeichen voranstellen, gefolgt von den für Hexadezimalzahlen üblichen Darstellungssymbolen `0...9`, `A`, `B`, `C`, `D`, `E`, `F`. Natürlich gilt auch bei Pascal, daß eine Hexadezimalzahl und eine Dezimalzahl letztendlich für das gleiche Bitmuster im Rechner herhalten können, mithin also unterschiedliche Darstellungen dieses Bitmusters repräsentieren.

Ein Beispiel für eine so dargestellte »Hexzahl« ist `$A12F`. Das dahinterstehende Bitmuster kann auch als dezimale Darstellung ausgedrückt werden, und dann erhalten Sie `41263`.

Fließkommazahlen

Für die Darstellung einer Fließkommakonstante gelten die Schreibweisen, wie Sie Ihnen sicherlich auch in ähnlicher Form aus dem Alltag geläufig sind.

Einzig bei der Darstellung von Exponentialzahlen zur Basis 10 mußte ein Kompromiß gemacht werden, um diese Zahlen darstellen zu können. Statt dem Exponenten 10 wird für ihn die Ersatzdarstellung `e` oder `E` genommen.

Boolsche Werte

Die *Wahrheitswerte* werden durch die beiden Symbole `True` für »wahr« und `False` für »falsch« repräsentiert. Sie können Sie überall da einsetzen, wo solche logischen Ausdrücke entstehen bzw. zugelassen sind.

Zeichen

Eine *Zeichenkonstante* wird innerhalb von einfachen Anführungszeichen dargestellt, so als wenn es sich um eine Zeichenkette handelt.

Trotzdem wird in Pascal in manchen Situationen streng zwischen diesen beiden Datentypen unterschieden!

Wollen Sie einzelne Zeichen mit ihrer ASCII-Kodierung angeben, was besonders dann der Fall ist, wenn Sie nicht druckbare Zeichen einsetzen wollen, müssen Sie das #-Zeichen gefolgt vom ASCII-Wert des darzustellenden Zeichens angeben:

```
#13 { Carriage Return }
#10 { Line Feed }
```

Textkonstanten

Unter Pascal wird eine *Textkonstante* durch ein einfaches Anführungszeichen eingeleitet und beendet. Alle zwischen diesen beiden Anführungszeichen liegenden Zeichen werden vom Compiler nicht interpretiert und gelten als Textkonstante. Stehen mehrere solcher Zeichenketten direkt hintereinander, dann werden sie vom Compiler zu einer einzigen zusammengefaßt.

Leere Zeichenketten bestehen aus zwei direkt aufeinanderfolgenden Hochkommas, wenn Sie das Hochkomma innerhalb eines Textes verwenden wollen, geben Sie es einfach zweimal hintereinander an.

2.2.2 Symbolische Konstante

Die Benutzung literaler Konstanten hat vor allem zwei gravierende Nachteile:

- Bei Änderungen müssen alle Zeilen des Programms abgesucht werden, ob dort die zu ändernden Werte stehen. Bei der nun folgenden Anpassung können Fehler eintreten, die ein vorher funktionsfähiges Programm ruinieren. Und wer sagt denn, daß 3.141 immer unsere Kreiskonstante ist? Ob die Zahl überhaupt von einer Änderung betroffen ist, müssen Sie also auch noch (richtig) entscheiden.

- In Programmtexten sind »magische Zahlen« zu finden. Das sind literale Konstanten – Zahlen und Zeichenketten – die im Programm genutzt werden, deren Bedeutung aber niemandem mehr so recht klar ist. Warum hat der Autor dort den Wert 3.141 benutzt? Welches 3.141 ist damit denn gemeint? Programme sind so viel schlechter les- und verstehbar.

Das Konzept der symbolischen Konstante

Um diese beiden Probleme – Lesbarkeit und Wartbarkeit – leichter in den Griff zu bekommen, bietet Pascal die Möglichkeit, sogenannte *symbolische Konstanten* zu definieren. Dabei handelt es sich um von Ihnen frei gewählte Worte, die mit einem Wert (und dem dazugehörigen Datentyp) assoziiert werden.

Im Gegensatz zu einer Variablen ist diese Wertzuweisung nicht mehr änderbar. Überall dort, wo Sie die symbolische Konstante einsetzen, wird der Compiler bei der Programmübersetzung stellvertretend für Sie den gewünschten festen Wert eintragen. Die Vorteile sind offensichtlich:

- Wenn sich der Definitionswert einer Konstante einmal ändert, müssen Sie nur an einer zentralen Stelle die Korrektur vornehmen. Eben an der Stelle der Definition. Beispielsweise im Zusammenhang mit Steuersätzen kann das recht praktisch sein.
- Im Programmtext erschließt sie die Bedeutung einer Konstanten sehr viel einfacher. Man muß nicht mehr raten, ob der gerade benutzte Wert `1` ein Korrekturfaktor ist, oder ein Aufschlag, oder ein Basiswert. Das ergibt sich aus dem Namen, den Sie für diese Konstante vergeben haben.

Definition einer symbolischen Konstanten

Die einfachste Konstantenform werden Sie hier kennenlernen. Dabei werden Sie nach dem Beginn des Programms, also nach `program` oder nach dem Beginn einer Prozedur das Schlüsselwort `const` folgen lassen. Dann kommt ein Name für die Konstante und ein Gleichheitszeichen, dem der mit der Konstanten assoziierte Wert folgt.

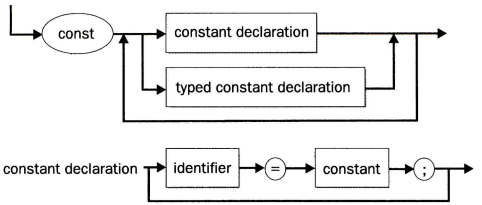


Bild 2.3: Syntax der Konstantendefinition

Abgeschlossen wird eine Definition durch das Semikolon. Sie können nun eine Konstante nach der anderen definieren, ein Vorgang der immer weitergehen kann, bis beispielsweise der Teil für die Variablenvereinbarungen mit seinem Wort `var` beginnt.

```
 1 program Konstanten;
 2
 3 const
 4 PI = 3.141; { Fließkommakonstante }
 5     NAME = 'ucuber'; { Zeichenkettenkonstante }
 6     JAHR = 1995; { Ganzzahlkonstante }
 7     WAHL = 'j'; { Zeichenkonstante }
 8
 9 begin { hier beginnt das Programm }
10     WriteLn('Kreiszahl ', PI);
11     WriteLn('Benutzername ', NAME);
12     WriteLn('Erscheinungsjahr', JAHR);
13     WriteLn('Gewähltes Zeichen ', WAHL);
14 end.
```

Listing 2.1: Nutzung von Konstanten (🖫 2_1.PAS)

2.2.3 Variablen und ihre Verwendung

Um den Speicher möglichst optimal verwalten zu können, muß Pascal schon gleich zur Übersetzungszeit wissen, wie eine Variable heißen soll und von welchem Typ sie ist.

Daher müssen Sie in Ihren Programmen vor der Benutzung einer Variablen diese auch *deklariert* haben. Ein positiver Nebeneffekt ist der, daß Tippfehler bei Variablennamen nicht mehr vorkommen können bzw. sofort bemerkt werden.

Denn aus Sicht von Pascal sind `Wariable` und `Variable` eben zwei unterschiedliche Namen. Und wenn einer davon nicht vorher bekanntgemacht wurde, dann muß hier wohl ein Fehler vorliegen, oder? Der Unterschied zwischen der Groß- und Kleinschreibung fällt allerdings nicht auf, da Pascal hier keine Unterscheidung trifft.

Die Variablenvereinbarung

Um eine Variable zu vereinbaren, gibt es im Programm und den Funktionen und Prozeduren genau dafür vorgesehene Stellen. Hier, und nur hier, dürfen neue Variablen vereinbart werden. An allen anderen Orten kassieren Sie eine Fehlermeldung.

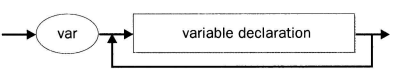


Bild 2.4: Beginn Variablenvereinbarung

Sie muß ziemlich zu Beginn des Programms in dem Bereich zwischen `program` und `begin` nach dem Schlüsselwort `var` vorgenommen werden.

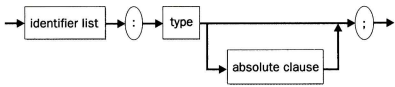


Bild 2.5: Namensvereinbarungen

Danach geben Sie eine Liste von Namen an, die durch Kommata getrennt werden. Dies sind die später zur Verfügung stehenden Variablennamen. Abgeschlossen wird die Liste durch einen Doppelpunkt, und dann folgt die Beschreibung des mit diesen Namen verbundenen Datentyps.

Dabei kann es sich um ganz einfache Typen wie ein Zeichen (`char`), eine Ganzzahl (`integer`), eine Fließkommazahl (`real`) oder eine Zeichenkette (`string`) handeln. Oder um komplexere Typen, wie ein Array, und sogar um von Ihnen definierte Typen.

```
program Programmname;
var
    name : Datentyp;
    name, name, name : Datentyp;
    name : Typ
    ...
begin
    ...
end.
```

Wichtig ist, daß die Variable auch den für ihre spätere Verwendung passenden Datentyp hat, da Sie in Pascal auf eine Variable des Typs Ganzzahl nicht einfach einen Wert des Typs Fließkommazahl übertragen dürfen. Das gibt sofort eine Fehlermeldung bei der Übersetzung.

Wenn Sie mit einem Satz Variablenvereinbarungen fertig sind, können Sie in der nächsten Zeile die nächste Variable oder Variablengruppe definieren. Das ist solange möglich, bis eine andere Programmeinheit eingeleitet wird.

Variablen einen Wert zuweisen

Nun müssen wir Ihnen noch zeigen, wie Sie einen Wert in eine Variable hinein bekommen. Es gibt da mehrere Möglichkeiten:

- Durch eine Wertzuweisung:
  `variable := wert;`
- Durch einen Prozeduraufruf, bei dem die Variable mitgegeben wird:
  `Prozedur(variable);`
- Durch Entgegennahme eines Funktionsergebnisses:
  `variable := Funktion;`

Ein Beispiel

Das folgende Programmbeispiel zeigt Ihnen, wie mit Hilfe von Zuweisungen Werte in Variablen eingelesen und dann wieder ausgegeben werden. Beachten Sie dabei, daß wir im Gegensatz zu dem Beispiel mit den Konstanten die Variablen mehrfach mit unterschiedlichen Werte belegen können:

```
 1 program Variablen;
 2
 3 var
 4     pi : real; { Fließkommazahl }
 5     name : string; { Zeichenkette }
 6     jahr : integer; { Ganzzahl }
 7     wahl : char; { Zeichen }
 8     pi := 3.141;
 9 begin { hier beginnt das Programm }
10     name := 'ucuber';
11     jahr := 1995;
12     wahl := 'j';
13
14     WriteLn('Kreiszahl ', pi);
15     WriteLn('Benutzername ', name);
16     WriteLn('Erscheinungsjahr ', jahr);
17     WriteLn('Gewähltes Zeichen ', wahl);
```

```
18
19      pi := 3.1415;
20      name := 'hwenzel';
21      jahr := 1996;
22      wahl := 'n';
23
24      WriteLn('Kreiszahl ', pi);
25      WriteLn('Benutzername ', name);
26      WriteLn('Erscheinungsjahr ', jahr);
27      WriteLn('Gewähltes Zeichen ', wahl);
28
29 end.
```

Listing 2.2: Beispiel für die Verwendung von Variablen (2_2.PAS)

Über die Lebensdauer und die Zugreifbarkeit auf Variablen erfahren Sie mehr im Kapitel über die Funktionen und Prozeduren.

2.3 Die Eigenschaften ordinaler Datentypen

Die ordinalen Typen sind in Pascal die Zeichen, – denken Sie an die Abbildung der Symbole auf die 256 ASCII-Werte – die Ganzzahlen und die logischen Werte.

2.3.1 Behandlung ordinaler Datentypen

Bevor wir Ihnen diese Datentypen näher präsentieren, möchten wir Sie noch mit einer Gruppe von Funktionen bzw. Operatoren bekannt machen, die speziell für den Einsatz mit Ordinalzahlen geschaffen wurden.

Die Ordinalroutinen

Tabelle 2.2: Ordinalroutinen

Routine	Einsatz
Ord(Element)	Liefert die Ordnungsnummer des angegebenen Elements.
Succ(Element)	Liefert den Nachfolger des Arguments, bzw. einen Fehler bei Erreichen des letzten Elements.
Pred(Element)	Liefert den Vorgänger des Arguments, bzw. einen Fehler beim Unterschreiten des ersten Elements.
Low(Element)	Liefert das erste Element des Wertebereichs, dem das angegebene Element angehört.
High(Element)	Liefert das letzte Element des Wertebereichs, dem das angegebene Element angehört.

Bestimmung der ordinalen Umgebung eines Wertes

Das folgende Beispielprogramm nutzt diese Ordinalroutinen aus, um eine Ganzzahl zu bewerten.

Bitte beachten Sie, wie wir in diesem Beispiel 5 Variablen vom Typ Ganzzahl deklarieren und eine davon mit der in diesem Kapitel noch behandelten Einlesefunktion `ReadLn()` mit einem Wert belegen.

```
 1 program TestOrdinal;
 2
 3 var
 4    h, l, e, p, s: integer;
 5
 6 begin
 7    { Eingabeaufforderung }
 8    Write('Eine Eingabe einer Ganzzahl bitte: ');
 9    ReadLn(e);
10
```

```
11    { Bestimmung von High und Low,
12      Ergebnis ist von der Eingabe unabhängig }
13    h := High(e);
14    l := Low(e);
15
16    { Bestimmung von Nachfolger und Vorgänger,
17      Ergebnis ist von der Eingabe abhängig,
18      keine Fehlerbehandlung!!! }
19    p:= Pred(e);
20    s := Succ(e);
21
22    { Ausgabe der Ergebnisse }
23    WriteLn('Höchster möglicher Wert:    ', h);
24    WriteLn('Niedrigster möglicher Wert: ', l);
25    WriteLn('Vorgängerwert:  ', p);
26    WriteLn('Nachfolgerwert: ', s);
27 end.
```

Listing 2.3: Beispiel für ordinale Datentypen (🖫 2_3.PAS)

Nachdem die Eingabe vom Benutzer gelesen wurde, werden auf diesen Wert die Ordinalfunktionen angewandt und ihre Rückgabewerte in Zuweisungen an andere Variablen gebunden. Letztendlich erfolgt dann die Ausgabe der Ergebnisse mit dem schon bekannten `WriteLn()`.

Die Bestimmung der größten und kleinsten darstellbaren Zahl ist nur vom Datentyp abhängig, nicht von der eingegebenen Zahl. Hier wird immer das gleiche Wertepaar (`-32768, 32767`) auftreten.

Anders sieht es dagegen bei dem Nachfolger und dem Vorgänger der eingegebenen Zahl aus. Je nachdem, ob sich die Eingabe noch im gültigen Zahlenbereich befindet, werden Sie unterschiedliche Zahlen erhalten, die aber für eine bestimmte Zahl immer die gleichen sein werden.

ASCII-Konvertierung

Auch das folgende Beispiel nutzt die Eigenschaften von ordinalen Datentypen (`integer` für Ganzzahlen, `char` für Buchstaben nach dem ASCII-Zeichensatz), um zu einem vom Benutzer eingegebenen ASCII-Zeichen die passende Ordnungsnummer zu ermitteln.

Diese Ordnungsnummern werden wieder mit der Funktion `Chr()` in die dazu passenden Zeichen zurückverwandelt und ausgegeben.

```
program TestCharOrdinal;

 var
   c, cv, cn: char;
   p, s, e: integer;

begin
   { Eingabe des Zeichens }
   Write('Eingabe eines Buchstabens: ');
   ReadLn(c);

   { Umwandlungen in die ASCII-Werte,
     Ermittlung von Vorgänger und Nachfolger }
   e := Ord(c);
   p := Pred(e);
   s := Succ(e);

   { Rückumwandlung der ASCII-Werte in die Zeichen }
   cv := Chr(p);
   cn := Chr(s);

   { Ausgabe der Ergebnisse }
   WriteLn('ASCII-Wert   :  ', e);
   WriteLn('Vorgängerwert:  ', p, ' ', cv);
   WriteLn('Nachfolgerwert: ', s, ' ', cn);
end.
```

Listing 2.4: Ein ASCII-Umwandler (2_4.PAS)

Tip: Die Zeichendarstellung der Ziffern 0...9 korrespondiert nicht mit ihrer ASCII-Ordnungsnummer. Wundern Sie sich also nicht, wenn Sie bei einer Eingabe des Ziffernsymbols `1` in diesem Programm für die numerischen Werte des Vorgängers und des Nachfolgers andere Ergebnisse erhalten als bei dem vorangehenden Programm, welches nicht mit ASCII-Werten, sondern mit »echten« Ganzzahlen operierte!

2.3.2 Zeichen

Durch ihre über die ASCII-Tabelle gegebene enge Verwandtschaft mit den Ganzzahlen werden einzelne Zeichen auch in einem eigenen ordinalen Datentyp verwaltet. Der Typ dieser Variablen heißt `char` und ist 8 Bit groß.

Zusätzlich zu `Ord()`, `Succ()`, `Pred()`, `High()` und `Low()` ist noch eine weitere Funktion namens `Chr()` definiert worden, mit der sich aus einem gegebenen ASCII-Wert das darzustellende Zeichen ermitteln läßt. Damit ist `Chr()` gewissermaßen eine Umkehrfunktion zu `Ord()`:

```
Ord('a') = Chr(97).
```

2.3.3 Ganzzahlen

Ursprünglich gibt es in Pascal nur einen einzigen Datentyp für Ganzzahlen, nämlich `integer`. Auf einem PC ist er so groß, daß Zahlen zwischen -32.768 und 32.767 aufgenommen werden können. Dazu reichen 2 Byte Speicherplatz aus. Für die meisten Aufgaben in Programmen mag das reichen. Allerdings hat dieser Datentyp auch seine Nachteile.

Grenzprobleme

Immer dann, wenn Sie Werte benutzen möchten, die größer oder kleiner als die angegebenen Grenzwerte sind, kommt es mit `integer` zu Problemen. Handelt es sich um eine direkte Wertezuweisung,

spielt der Compiler schon nicht mit, daß heißt, Sie können einer Variablen vom `integer`-Typ keinen Wert `50000` zuweisen!

Problematischer wird es, wenn Sie im Rahmen von Berechnungen die Grenzen des `integer`-Typs verlassen. Dann gibt es nämlich keine Fehlermeldung, sondern das Programm läuft einfach weiter, genau wie es bei dem folgenden Beispielprogramm ist:

```
 1 program Ueberlauf;
 2
 3 var
 4    i: integer;
 5
 6 begin
 7      while i < 50000 do
 8            begin
 9                 i:= i + 1000;
10                 WriteLn(i);
11            end;
12 end.
```

Listing 2.5: Ein fehlerhaftes Laufzeitverhalten durch Ganzzahl-Überlauf (🖫 2_5.PAS)

In dem Programm wird eine `while`-Schleife abgearbeitet, die eigentlich dann stoppen sollte, wenn `i` einen Wert größer oder gleich `50000` erreicht hat.

Leider passiert das nicht, Sie erhalten als Programmausgabe eine ewige Folge von negativen und positiven Werten, und das Programm wird nie zu einem Ende kommen!

Was hier passiert, ist daß das Bitmuster, welches die Zahl intern repräsentiert, beim Überschreiten des Wertebereiches der Variablen `i` einfach abgeschnitten und auf 16 Bit angepaßt wird. So wird aus `1000000000000000` durch die Anpassung `0000000000000000`. Und das bedeutet, daß aus einem anderen Wert der Wert Null »erzeugt« wurde.

Durch die Besonderheiten, wie negative Zahlen in Computern dargestellt werden, wird das Programm immer in einem *Zahlenkreis* herumlaufen.

Negative Ganzzahldarstellung

Wenn Sie eine negative Zahl mit Hilfe von Ganzzahlwerten darstellen, muß der Computer eine ganz besondere Repräsentation des dazugehörigen Bitmusters auswählen.

Die Darstellung von vorzeichenbehafteten Zahlen erfolgt intern im *Zweierkomplement,* bei dem für negative Zahlendarstellung ein besonderes Bitmuster erzeugt wird. Die zur Verfügung stehenden Bitplätze der Zahl werden so aufgeteilt, daß das erste Bit für das Vorzeichen reserviert wird (0 – positiv, 1 – negativ). Die verbleibenden Stellen werden für die Aufnahme des Zahlenbitmusters genommen.

Die Umwandlung in die passende Darstellung erfolgt normalerweise automatisch und ohne Ihr Zutun. Manchmal, wie im obigen Programmbeispiel, allerdings eher unerwünschterweise!

Ein Beispiel soll daher den Vorgang erläutern: Um die Zahl -3 intern als Bitmuster darzustellen, geht der Compiler nach folgenden Schritten vor (aus Gründen der Übersichtlichkeit benutzen wir eine 8 Bit-Zahl):

- Darstellung des Bitmusters von 3: `00000011`
- Alle Bits werden »umgekehrt«: `11111100`
- Es wird der Wert 1 dazu addiert: `11111100 + 00000001`
- Die Darstellung von -3 ist dann: `11111101`

Das ist doch ganz einfach, oder? Um aus dieser Form des Bitmusters die Zahl wieder zu erhalten, müssen die obigen Schritte nur in der umgekehrten Reihenfolge angewandt werden. Das Kriterium, ob das überhaupt nötig ist, ist das erste Bit. Wenn es auf `1` steht, dann muß die Umwandlung wohl stattfinden, ansonsten kann das Bitmuster direkt genommen werden.

Erweiterung der Ganzzahltypen

Borland hat unter anderem wegen der oben genannten Probleme die Ganzzahlen als Datentyp ergänzt. Diese verschiedenen Ganzzahldatentypen zeichnen sich durch die verschieden großen Wertebereiche aus. Allerdings gilt auch hier die Regel, daß eine Überschreitung der Wertegrenzen zur Laufzeit keine Fehlermeldung und damit letztendlich irrelaufende Programme zur Folge hat.

Die Ganzzahlen, welche nur für die Aufnahme positiver Werte vorgesehen sind, reservieren übrigens kein Vorzeichenbit und können daher in der gleichen Anzahl Bitstellen größere Zahlen darstellen. Mit den folgenden Formeln läßt sich die maximal und minimal mögliche Zahl eines Ganzzahltyps leicht errechnen:

Positive Ganzzahlen: $0..2^{Größe} - 1$

Negative Ganzzahlen: $2^{(Größe-1)}...0...2^{(Größe-1)}-1$

Tabelle 2.3: Ganzzahltypen

Typname	Wertebereich, Größe
shortint	-128..0..127; 8 Bit
byte	0..255; 8 Bit
integer	-32768..0..32767; 16 Bit
word	0..65535; 16 Bit
longint	-2147483648..0..2147483647; 32 Bit

Konstante und Zuweisungen

Wenn Sie eine Konstante der Ganzzahltypen anlegen, wird übrigens automatisch der kleinste passende Typ ausgewählt, bei Variablen bestimmen Sie durch die Typangabe, wieviel Bits benutzt und wie sie verwaltet werden sollen.

Die verschiedenen Ganzzahlen sind ohne Typüberprüfung zueinander zuweisungskompatibel. Das heißt, Sie können zur Laufzeit jederzeit eine `byte`-Variable auf eine `word`-Variable abbilden und umgekehrt. Sollten dabei Informationen verlorengehen, wird es ebenfalls keine Fehlermeldung geben!

Nur wenn die Compileroption `{$R+}` eingestellt ist, wird eine solche fehlerhafte Operation zur Laufzeit erkannt und mit einer Fehlermeldung quittiert.

2.3.4 Wahrheitswerte

Eine weitere Klasse der ordinalen Datentypen sind die logischen oder auch Wahrheitswerte des Typs `boolean`. Die möglichen Werte sind `false` für logisch falsche und `true` für logisch wahre Werte.

Intern gilt, daß jeder ganzzahlige Wert ungleich 0 als `true` betrachtet wird, wogegen der Wert 0 als `false` gilt. Die logischen Operatoren und die Vergleichsoperatoren liefern den Wert `1` für `true` und `0` für `false` zurück.

Regeln für boolsche Zahlen

Da die boolschen Datentypen ordinale Typen sind, gelten die folgenden Regeln:

```
False < True
Ord(False) = 0
Ord(True) = 1
Succ(False) = True
Pred(True) = False
```

Die möglichen vorhandenen Datentypen für die Aufnahme boolscher Werte sind so definiert, daß sie von der Größe her gut mit den entsprechenden Datentypen der Ganzzahlen zusammenpassen und die Typumwandlungen zwischen ihnen so erleichtert werden.

Tabelle 2.4: Boolsche Datentypen

Name	Größe
boolean	8 Bit
bytebool	8 Bit
wordbool	16 Bit
longbool	32 Bit

2.4 Reale Datentypen

Fließkommazahlen, auch reelle Zahlen genannt, sind keine ordinalen Typen. Sie gehorchen daher anderen Regeln, als die bisher beschriebenen Datentypen. Auch sind die Ordinalfunktionen nicht direkt auf sie anwendbar.

2.4.1 Datentypen

Zu den hier angegebenen Typen muß noch gesagt werden, daß Sie Ihnen nur zur Verfügung stehen, wenn Ihr Computer einen numerischen Coprozessor hat (Compileroption `{$N}`). Wenn dies nicht der Fall ist, müssen Sie auf `real` zurückgreifen.

Tabelle 2.5: Die Fließkommatypen

Typname	Wertebereich, Signifikanz, Größe
real	2.9E-39...1.7E38; 12 Stellen; 6 Byte
single	1.5E-45...3.4E38; 8 Stellen; 4 Byte
double	5.0E-324...1.7E308; 16 Stellen; 8 Byte
extended	1.9E-4951...1.1E4932; 20 Stellen; 10 Byte
comp	(-2E63)+1...(2E63)-1; 20 Stellen; 8 Byte

2.4.2 Fehlerquellen

Während Sie bei den Ganzzahlen die Problematik des »Zahlenkreises« kennenlernten, gibt es für die Fließkommatypen eine andere Quelle von Verdruß. Denn sie sind nicht mehr auf die einfache Weise intern dargestellt, wie das bei den Ganzzahlen der Fall ist.

Überlauf, Unterlauf

Zuerst einmal zum Überlauf: Hier erhalten Sie einen Laufzeitfehler; das Programm wird abgebrochen, wenn Sie die gültige Wertegrenze überschreiten. Gleiches passiert, wenn Sie einen Unterlauf während einer Berechnung verursachen.

Exaktheit

Das für Fließkommatypen typische Problem ist die Frage der Exaktheit der Zahlendarstellung. Denn, wie Sie der Tabelle der Fließkommatypen entnehmen können, werden Ihnen geradezu riesige Wertebereiche zur Verfügung gestellt. Zahlen mit bis zu 4.900 Stellen sind – theoretisch – möglich. Der Haken dabei ist, daß das Darstellungsformat es gar nicht erlaubt, alle diese 4.900 Stellen einzeln und exakt anzugeben.

Es werden nur eine bestimmte Anzahl dieser Stellen gespeichert, und alles was darüber hinausgeht wird durch die Ziffer 0 ersetzt! Die gespeicherten Ziffern der Zahl sind dann die signifikanten Stellen.

Während also die reellen Zahlen sehr große Werte annehmen können, ist es mit der Anzahl der signifikanten Stellen nicht so weit her. Das bedeutet, daß bei einer Zahl mit 52 Stellen im günstigsten Fall höchstens die ersten 20 Stellen signifikant sind. Bei naturwissenschaftlichen Berechnungen ist das meist nicht so tragisch, bei finanzmathematischen oder ingenieursmäßigen Berechnungen dagegen kann sich diese gerade auch bei Kettenrechnungen akkumulierte Ungenauigkeit zu einem ziemlichen Problem ausweiten!

2.5 Zeichenketten

Zeichenketten haben Sie schon im ersten Kapitel kennengelernt. Viel mehr gibt es zu den Standardzeichenketten fast nicht zu sagen.

2.5.1 Zeichenketten sind nicht gleich Zeichenketten!

Eine Besonderheit in Borland Pascal 7.0 sind die Zeichenketten, die nach C/C++-Konventionen funktionieren. Sie wurden aus Gründen der Kompatibilität zum Windows-API als ein weiterer Zeichenkettentyp eingeführt.

Zeiger auf Zeichenketten

Neben den klassischen Pascal-Strings, die eine maximale Länge von 255 Zeichen erreichen dürfen und die keine besondere Endekennung haben, gibt es nun den Typ `PChar`, der einen Verweis auf eine Speicherstelle *(Zeiger)* repräsentiert. Dieser Zeiger verweist auf einen beliebig langen, mit einem besonderen Zeichen (`'\0'`) terminierten String im C-Stil. Außerdem gilt für einen String im C-Stil, daß er zuerst nichts weiter als ein Zeiger ist und erst einmal mit einer gültigen Zeichenfolge oder anderem, entsprechend vorbereiteten Speicherplatz verbunden werden muß.

Wir werden in diesem Buch nicht weiter auf diese Erweiterung eingehen und möchten Sie nur darauf hinweisen, daß beide Typen zueinander unverträglich sind und mit Hilfe spezieller Funktionen in der Bibliothek für Zeichenkettenfunktionen (`unit String`) ineinander umgewandelt werden können. Für weitere Informationen darüber wenden Sie sich bitte an die Online Hilfe unter den Stichworten *Routinen zur Verwendung von Strings (null-terminiert)* und *Routinen zur Verwendung von Strings (Pascal-Stil).*

2.5.2 Der Datentyp string

Wir werden in diesem Buch nur den originalen Datentyp namens `string` benutzen, der für Zeichenketten bis maximal 255 Zeichen geeignet ist. Während eine Zeichenkettenkonstante recht einfach aufgebaut ist, können Sie für Zeichenkettenvariablen einige Varianten der Vereinbarung einsetzen.

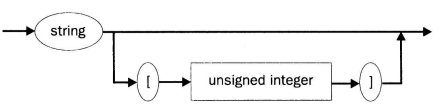


Bild 2.6: Syntax des string-Typs

Vereinbarungsvarianten

Eine Zeichenkette im Sinne von Pascal wird wie folgt vereinbart:

```
var
    s: string;
    Name: string[30];
    Ort: string[25];
    PLZ: string[5];
```

Eine Erweiterung des `string`-Typs ermöglicht schon zu Beginn eine Angabe über die Größe des zur Verfügung stehenden Speicherplatzes. Allerdings gilt auch hier, daß die maximale Stringlänge für die Pascal-Strings nur zwischen 1 und 255 liegen kann.

Zu beachten ist auch, daß es Probleme geben kann, wenn Sie den Inhalt eines Strings mit `string[5]` auf eine Stringvariable mit `string[10]` zuweisen wollen. Denn aus der Sicht von Pascal handelt es sich dabei um unterschiedliche Datentypen!

2.5.3 Stringfunktionen

Abgesehen von der Einschränkung mit den 255 Zeichen und der Tatsache, daß `string[3]` und `string[4]` unterschiedliche Datentypen sind, können Sie mit Zeichenketten in Pascal recht bequem arbeiten.

Stringfunktionen im Pascal-Stil

Letzteres gilt vor allem auch wegen der Vielzahl an vordefinierten Funktionen, deren Beschreibung Sie in der Hilfe finden können *(Routinen zur Verwendung von Strings (Pascal-Stil))*. Einige wichtige Funktionen zeigt Ihnen die Tabelle auf der folgenden Seite:

Tabelle 2.6: Zeichenkettenfunktionen

Funktion	Einsatz
Length(s)	Liefert die Länge eines Strings
Concat(s1, s2)	Liefert die verketteten Strings zurück
Copy(s,. p, l)	Liefert als Ergebnis l Zeichen aus s ab der Position p
Pos(s1, s2)	Liefert die Position, an der s1 in s2 zum ersten Mal vorkommt
Insert(s1, s2, p)	Fügt in s2 den String s1 ab der Position p ein
Delete(s, p, l)	Entfernt in s ab der Stelle p die Anzahl von l Zeichen
Str(v, s)	Verwandelt den numerischen Wert v in eine Zeichenkette
Val(s, v. p)	Verwandelt eine Zeichenkette s in einen numerischen Wert in v. In p steht, ob die Umwandlung erfolgreich war. Wenn das so ist, dann steht hier der Wert 0

Ein Beispiel

Das folgende Beispiel zeigt Ihnen, wie mit den Stringfunktionen für Pascal-Strings gearbeitet werden kann.

Im Programm sind drei Stringvariablen definiert worden. In diesem Fall ohne Angabe einer festen Länge. Die Zuweisung des dort stehenden Inhalts erfolgt ganz unkompliziert in Zeile 7 und 8. Der Inhalt von `s3` wird durch *Konkatenation* in Zeile 10 erzeugt.

Die Ausgabe des neuen Strings erfolgt gleichzeitig mit der Bestimmung der neuen Gesamtlänge des Inhalts von `s3`. Danach lassen wir die Position ausgeben, an der das Wort »`hey`« – also der Inhalt von `s2` – in `s3` zum ersten Mal vorkommt. Genau die gleiche Funktion machen wir uns zunutze, um in Zeile 15 in den String `s3` ein Komma nach dem »`Aloha`« einfügen zu lassen.

```
 1 program Zeichenkette;
 2
 3 var
 4    s1, s2, s3 : string;
 5
 6 begin
 7      s1 := 'Aloha ';
 8      s2 := 'hey!';
 9
10      s3 := Concat(s1, s2);
11      WriteLn(Length(s3), '---', s3);
12
13      WriteLn(Pos(s2, s3));
14
15      Insert(',', s3, Pos(s2, s3)-1);
16      WriteLn(s3);
17 end.
```

Listing 2.6: Beispiel für die Verwendung von Stringfunktionen (2_6.PAS)

2.6 Kleines Intermezzo: Dateneingabe

Befassen wir uns nun mit den *Eingabefunktionen*. Denn natürlich muß ein Programm auch Benutzereingaben entgegennehmen können.

Mittlerweile sind die Benutzer allerdings die komfortabelsten Schnittstellen gewohnt, mit Mausklick, Menüs usw. Wie Sie sich sicherlich denken können, verlangt die Programmierung und Gestaltung solch einer Benutzerschnittstelle einiges an Aufwand.

Wir beschränken uns auf die beiden Funktionen, mit denen unter anderem von der Tastatur gelesen werden kann. Als Ziel der Leseoperation muß eine Variable angegeben werden, die die Eingabe aufnimmt.

2.6.1 Eingabefunktionen Read() und ReadLn()

Das Lesen von der Tastatur wird wahlweise mit `Read()` oder `ReadLn()` realisiert. Verbunden mit dem *Standardeingabekanal* nimmt Ihr Programm so die Zeichen von der Tastatur entgegen und legt sie in Variablen ab.

```
 1 program EingabeTest;
 2
 3 var
 4     eingabe: string;
 5
 6 begin
 7     Write('Ihre Eingabe: ');
 8     ReadLn(eingabe);
 9
10     WriteLn('Sie gaben ein: ', eingabe, ' !!!');
11 end.
```

Listing 2.7: Text- und Zahleneingabe mit ReadLn() (💾 **2_7.PAS**)

Die Variable heißt `eingabe` und soll eine Zeichenkette – Datentyp `string` – aufnehmen. Nach der Variablendeklaration hinter `var` und dem Beginn des Programms nach `begin` folgt die Eingabeaufforderung. Und dann kommt die Funktion `ReadLn()`,die alles, was Sie eintippen, in der Variablen `eingabe` sammelt. Danach gibt das Programm diese Eingabe – mit einem Kommentar versehen – wieder aus.

2.6.2 Einlesen von Zahlen

Auf ähnliche Art können Sie auch Zahlen einlesen. Statt der alten Variablenvereinbarung

```
var
    eingabe: string;
```

müssen Sie nur die neue Variablenvereinbarung für Ganzzahlen

```
var
    eingabe: integer;
```

oder Realzahlen

```
var
    eingabe: real;
```

einsetzen. Der Rest des Programms kann unverändert bleiben, da die Funktion `ReadLn()` aus dem Typ der übergebenen Variablen schließen kann, welche Eingabeform genommen werden soll.

Lassen Sie sich übrigens nicht davon täuschen, daß Sie auch bei der Verwendung einer Stringvariablen Zahlen eingeben können. Die werden nämlich in Zeichenketten umgewandelt. Rechnen können Sie damit ohne besondere Vorkehrungen nicht mehr!

2.6.3 Unterschiede zwischen Read() und ReadLn()

Der Unterschied zwischen `ReadLn()` und `Read()` beim Einlesen ist der, daß `ReadLn()` die Zeilenendekennung mit verbraucht, während `Read()` die Zeilenendekennung im Tastaturpuffer stehen läßt. Der Unterschied scheint akademisch zu sein, kann aber bei bestimmten Fällen der Benutzereingabe von Bedeutung sein.

Mit Pascal können Sie rechnen 3

3.1 Ausdruck, Anweisung, Anweisungsblock 72

3.2 Operatoren 76

3.3 Regeln für die Typumwandlung 90

Nachdem Sie im letzten Kapitel die Datentypen und die daraus abgeleiteten Konstanten und Variablen kennengelernt haben, wird es nun Zeit, Ihnen zu zeigen, wie Sie mit ihnen rechnen können.

Dazu benötigen Sie Kenntnisse über die *Operatoren,* die für verschiedene Datentypen anwendbar sind. Machen Sie sich dabei auf eine Überraschung gefaßt! Denn Pascal läßt es nicht zu, daß Sie in einem Ausdruck beliebige Datentypen miteinander vermischen. Wenn der Compiler keine Möglichkeit sieht, die einzelnen Datentypen sinnvoll aufeinander abzubilden, hagelt es Fehlermeldungen.

3.1 Ausdruck, Anweisung, Anweisungsblock

Bisher haben wir die Begriffe *Ausdruck* und *Anweisung* bzw. *Anweisungsblock* genutzt, ohne uns weiter über deren Bedeutung auszulassen. Es ist also an der Zeit, die Begriffe einmal genauer zu definieren. Und das bietet sich im Zusammenhang mit den Operatoren an.

3.1.1 Der Ausdruck

Ausdrücke stehen neben den Kommandos der Programmiersprache, mit denen sie gemeinsam Anweisungen ergeben.

Ausdruckswerte

Ein Ausdruck besteht normalerweise aus einem oder mehreren *Operatoren* und *Operanden.* Die Operatoren verknüpfen die Operanden auf eine ihnen zugeordnete Art und Weise.

Ein Beispiel für einen Ausdruck ist die Addition, wo zwei Operanden über den Plus-Operator verknüpft werden. Handelt es sich um numerische Operanden, wird die Addition ausgeführt, die Summe gebildet und als Ausdruckswert zurückgegeben. Dieser Ausdruckswert kann dann beispielsweise in einer Zuweisung genutzt werden oder in anderen Ausdrücken weiter verarbeitet werden.

```
erg := 3 + 5;
erg = A / 55;
```

Die Operanden eines Ausdrucks können sowohl Variableninhalte, als auch Konstanten oder Rückgabewerte von Funktionsaufrufen sein.

Der einfachste Ausdruck ist übrigens eine einsame Konstante! Ihr Ausdruckswert ist gleich dem mit ihr verbundenen literalen Konstantenwert.

Verschachtelung

Es ist nicht ungewöhnlich, daß sich ein Ausdruck aus mehreren Einzelausdrücken zusammensetzt. In dem Fall müssen Regeln aufgestellt werden, die klären, welcher Teilausdruck als erster auszuführen ist.

Ein Beispiel aus der Welt der Schulmathematik ist eine Additionskette. Der Ausdruck `A + B + C` kann im Prinzip in beliebiger Folge abgearbeitet werden, es muß immer das gleiche Ergebnis dabei herauskommen. Für die Programmiersprache Pascal ist dagegen festgelegt, daß dieser Ausdruck von links nach rechts ausgewertet wird. Also wird zuerst `A + B` errechnet und dieser Ausdruckswert dann mit `C` verknüpft.

Seiteneffekte

Interessant wird es, wenn statt feststehender Größen oder Variablen Funktionen und ihre Rückgabewerte als Operanden eingesetzt werden. In dem Fall von `F1() + F2() + F3()` bedeutet das nämlich, daß zuerst die Funktion `F1()` abgearbeitet wird, dann `F2()` und zuletzt `F3()`. Sollten die Funktionen unabhängig voneinander sein, macht das nichts. Aber stellen Sie sich einmal vor, alle drei Funktionen würden auf einen gemeinsamen Wert zugreifen und diesen beim Aufruf verändern.

Dann kann es wohl vorkommen, daß das Gesamtprogramm sich nicht so verhält, wie es geplant war. Diesen Effekt, daß ein Funktionsaufruf etwas verändert, was wiederum das Verhalten einer anderen Funktion beeinflußt, nennt man *Seiteneffekt.* Manchmal ist er gewünscht, doch oft eher eine programmiertechnische Panne.

Prioritäten

Operatoren haben *Prioritäten.* In einer Ausdruckskette entscheiden diese Prioritäten darüber, welcher Teilausdruck zuerst abgearbeitet wird. Auch hier zeigt ein Beispiel aus der täglichen Mathematik wieder, was gemeint ist.

In `A + B * C` sieht die Sache so aus, daß die Multiplikation gegenüber der Addition die höhere Priorität hat. Daher wird erst der Teilausdruck `B * C` ausgewertet und dessen Ausdruckswert mit `A` verknüpft.

Natürlich kann der Ausdruck durch den Einsatz von Klammern, die auch in Pascal die höchste Priorität haben, anders ausgewertet werden. Daraus folgt dann, daß `(A + B) * C` ungleich `A + B * C` ist.

Tabelle 3.1: Operatorenreihenfolge

Operatoren	**Analogie zur Arithmetik**
( )	Klammerung
@ not -	Vorzeichen
* / div mod and shl shr	Multiplikation
+ - or xor	Addition
= < > =< >= <> in	Vergleich

Diese Prioritätsregeln gelten für alle Operatoren, die Pascal anbietet, und müssen von Ihnen bei der Formulierung der Ausdrücke berücksichtigt werden.

3.1.2 Die Anweisungen und der Anweisungsblock

Eine Anweisung ist eine vollständige Programmanweisung. Anweisungen bestehen aus Ausdrücken und Befehlsworten der Programmiersprache, die mit einem Semikolon abgeschlossen sind.

Wann setzte ich ein Semikolon?

Es gibt Fälle, bei denen scheinbar kein Semikolon stehen muß. Allerdings täuscht das, denn durch die Kombination mit anderen Elementen von Pascal ist doch eine Anweisung entstanden, die dann nur über mehrere Zeilen geht. Beispiele dafür sind der Anweisungblock oder die `if`-Verzweigung:

```
begin
    action
end;

if Bedingung then
    action;
```

In diesen Beispielen muß man die einzelnen Zeilen so lesen, als wenn sie eine einzige wären. Und dann steht das Semikolon wieder an der richtigen Stelle.

Die Zuweisung als Anweisung

Während beispielsweise eine Addition ein Ausdruck ist, wird sie durch die Kombination mit der Zuweisung und einem Semikolon Teil einer Anweisung.

```
erg := 55;
erg := a + b;
erg := Funktion();
```

Der Anweisungsblock

Aber auch der Aufruf einer Prozedur ist eine Anweisung oder eine Schleife oder Verzweigung. Nach der Sprachdefinition kann beispielsweise hinter dem `then` einer Verzweigung eine Anweisung stehen – oder ein Anweisungsblock. Dann nämlich, wenn für diesen Fall mehrere Anweisungen statt einer abgearbeitet werden sollen.

Ein Anweisungsblock beginnt mit `begin` und schließt mit `end`. Alle dazwischen stehenden Anweisungen werden jetzt bezüglich des `then`-Zweigs als eine Anweisung betrachtet.

Und natürlich dürfen solche Anweisungsblöcke auch ineinander geschachtelt sein.

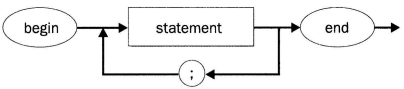


Bild 3.1: Syntax des Anweisungsblocks

3.2 Operatoren

Hier nun die *Operatoren,* welche Ihnen Pascal anbietet. Was folgt, sind die Operatoren, die zu den bisher vorgestellten Datentypen passen. Daneben sind noch einige andere Operatoren vorhanden, die sich vor allem auf die Nutzung der fortgeschrittenen Datentypen und der benutzerdefinierten Typen beziehen. Diese Operatoren lernen Sie dann im Zusammenhang mit diesen Pascalkonzepten kennen.

Und wir möchten noch einmal darauf hinweisen, daß Pascal es nicht zuläßt, wenn zwei nicht zueinander passende Operanden verknüpft werden. Wollen Sie beispielsweise eine Zeichenkette und eine Ganzzahl miteinander verknüpfen, müssen Sie erst einen der beiden Partner in eine Darstellung umwandeln, die vom Typ her zum anderen Partner identisch ist.

3.2.1 Arithmetische Operatoren

Die arithmetischen Operatoren sind die gleichen, wie Sie sie auch schon aus der Schule kennen. Später zeigen wir, daß der Additionsoperator auch auf die Verknüpfung von Zeichenketten erweitert wurde.

Achtung beim Ausdruckswert

Eine Besonderheit, die Sie berücksichtigen müssen, ist der Ausdruckswerttyp. Denn natürlich erwarten Sie, daß die Ausdruckswerte von einem bestimmten Typ sind, der unter anderem von den Operanden mit bestimmt wird.

Während das beispielsweise bei der Addition der Fall ist, handelt es sich bei der Division um einen Sonderfall: Ihr Ausdruckswert ist immer vom Typ `real`! Das ist natürlich – in Anbetracht pascalscher Typstrenge – bei Zuweisungen auf Ergebnisvariablen zu berücksichtigen.

Tabelle 3.2: Arithmetische Operatoren

Operator	Operanden	Ergebnis	Operation
+	I, R	I, R	Addition
-	I, R	I, R	Subtraktion
*	I, R	I, R	Multiplikation
/	I, R	R	Division
DIV	I	I	ganze Division
MOD	I	I	Modulo

Die Kürzel I und R stehen für `integer` und `real` als Datentypen, die für die Operation erlaubt sind. Gleiches gilt für das Ausdrucksergebnis. Wenn Sie ganzzahlige Operanden mit Fließkommaoperanden mischen, wird der ganzzahlige Operand auf den Fließkommatyp angepaßt, und das Ergebnis ist ein `real`-Wert.

Division und Modulo

Die Operatoren `div` und `mod` sind Spezialfälle für die ganzzahlige Division. Mit `div` erhalten Sie den ganzzahligen Anteil der Division, und `mod` liefert Ihnen den Divisionsrest.

```
 1 program DivisionsTest;
 2
 3 var
 4     modulo, division : integer;
 5     erg : real;
 6
 7 begin
 8     erg := 5 / 3;
 9
10     division := 5 div 3;
11     modulo := 5 mod 3;
12
13     WriteLn('5 / 3 =  ', erg);
14     WriteLn('oder ', division, ' Rest ', modulo);
15 end.
```

Listing 3.1: Beispiele für die Varianten der Division (🖫 3_1.PAS)

Die Zeichenkettenaddition

Speziell definierte Operatoren für Zeichenketten gibt es in Pascal nicht. Es werden einige der schon bekannten Operatoren mit genutzt.

Da wäre zum einen die Addition zu nennen, mit der zwei Zeichenketten aneinander gebunden werden können. Natürlich müssen Sie dabei darauf achten, daß die erlaubte Gesamtlänge von maximal 255 Zeichen nicht überschritten wird.

```
1 program Namen;
2
3 var
4     vname, nname, name : string;
5
6 begin
7     Write('Ihr Nachname: ');
8     ReadLn(nname);
9
```

```
10      Write('Ihr Vorname: ');
11      ReadLn(vname);
12
13      name := vname + ' ' + nname;
14      WriteLn('Willkommen ', name);
15 end.
```

Listing 3.2: Stringaddition (3_2.PAS)

3.2.2 Vergleichsoperatoren

Eine der häufigsten Aufgaben in Programmen ist der Vergleich von Werten miteinander. Je nach Vergleichsergebnis entscheidet sich dann, wie das Programm weiter abläuft.

Die verschiedenen Testoperatoren haben leichte Ähnlichkeit mit den Zeichen aus der Mathematik, dargestellt mit den Beschränkungen des modernen Computerzeichensatzes.

Die Bedeutung der Operatoren

Vergleichsoperatoren dienen dem Vergleich zweier Werte. Damit können Sie genauso gut Zeichenketten vergleichen, wie Fließkommazahlen, Ganzzahlen, logische Werte usw.

Als Ergebnis erhalten Sie `true`, wenn der Vergleich ein wahres Ergebnis liefert, ansonsten `false`.

```
1 program Vergleiche;
2
3 var
4      erg : boolean;
5      vi : integer;
6      vr : real;
7
8 begin
9            WriteLn('Willi' = 'willi'); { Zeichenketten }
```

```
10      WriteLn(23 < 24); { numerische Werte }
11      WriteLn((1 + 1) = 2); { komplexer Ausdruck }
12
13      vi := 9;
14      vr := 9.0;
15
16      erg := vi = vr; { gemischter Vergleich }
17      WriteLn(erg);
18 end.
```

Listing 3.3: Vergleichsoperatoren (3_3.PAS)

Natürlich gilt bei den Vergleichsoperatoren der Fall, daß eine Variable, die den Ergebniswert des Ausdrucks aufnimmt vom richtigen Typ, also `boolean` ist, und daß die Operanden auch typkompatibel sind – wobei eine Ganzzahl beim Vergleich mit einer Fließkommazahl auf den Fließkommatyp angepaßt wird. In allen anderen Fällen erscheint gleich wieder eine Fehlermeldung vom Compiler. Denn in Pascal dürfen Sie eben nicht Äpfel mit Birnen vergleichen.

Tabelle 3.3: Vergleichsoperatoren

Operator	Bedeutung
=	Gleichheit
<>	Ungleichheit
<	Kleiner als
<=	Kleiner oder gleich
>	Größer als
>=	Größer oder gleich

Zeichenkettenvergleiche

Bei Vergleichen von Zeichenketten handelt es sich nicht um Vergleiche auf der Ebene der Zeichenzahl, sondern es werden die ASCII-Werte der Zeichen für die Vergleiche benutzt. Wenn man also testet,

ob 'willi' ungleich 'Willi' ist, wird man ein true erhalten, da der ASCII-Wert für das kleine w ein anderer ist als beim großen W.

Gleiches gilt, wenn Sie Vergleiche auf kleiner oder größer vornehmen. Hier werden immer die ASCII-Werte der einzelnen Zeichen miteinander verglichen, was bedeutet, daß ein Großbuchstabe immer einen kleineren ASCII-Wert hat als ein beliebiger Kleinbuchstabe.

Und so kommt es, daß der Vergleich 'Hamburg' < 'München' wahr ist, was allerdings nur für die ASCII-Werte der Buchstaben H und M gilt!

```
 1 program NamensVergleich;
 2
 3 var
 4     eingabe: string;
 5
 6 begin
 7     Write('Ihr Name: ');
 8     ReadLn(eingabe);
 9
10     WriteLn(eingabe = 'Anne');
11     WriteLn(eingabe < 'Anne');
12     WriteLn(eingabe > 'Anne');
13 end.
```

Listing 3.4: Stringvergleiche (3_4.PAS)

3.2.3 Logische (Boolsche) Operatoren

Die logischen Operatoren erwarten boolsche Operanden, also Ausdruckswerte true oder false, und liefern ihrerseits wieder true oder false als Ergebnis zurück. Somit lassen sich auch komplexere Abfragen und Vergleiche auf einfache Art realisieren.

Die Bedeutung der Operatoren

Insgesamt existieren vier verschiedene boolsche Operatoren. Das *Logische Nicht* (`not`) bedeutet eine Verneinung einer Aussage. Immer wenn die Aussage selber `true` ist, wird ihre Verneinung `false` ergeben und umgekehrt. Damit entspricht das *Logische Nicht* auch menschlichen Sprachgewohnheiten.

Tabelle 3.4: Boolsche Operatoren

Operator	Bedeutung
not	Logisches NICHT(Verneinung)
and	Logisches UND
or	Logisches ODER
xor	Logisches Exklusiv-Oder

Beim *Logischen Und* (`and`) ist die Gesamtaussage nur dann wahr, wenn beide Teilaussagen wahr sind. Ist eine von ihnen nicht wahr, dann liefert der Operator das Ergebnis `false` zurück.

Beim *Logischen Oder* (`or`) sieht die Sache so aus, daß das Ergebnis nur dann `false` ist, wenn beide Operanden `false` sind. Ist nur eine Teilaussage wahr, wird sofort die Gesamtaussage wahr.

Der dritte im Bunde ist das *Logische Exklusiv Oder* (`xor`). Bei ihm sieht die Sache so aus, daß das Ergebnis nur dann `true` wird, wenn wirklich nur einer der beiden Operanden selber `true` ist. Sind beide `true` oder `false`, ist das Ergebnis auch `false`. Damit kommt das `XOR` einem umgangssprachlichen »entweder oder« recht nahe.

A	B	not A	not B	A and B	A or B	A xor B
t	t	f	f	t	t	f
t	f	f	t	f	t	t
f	t	t	f	f	t	t
f	f	t	t	f	f	f

Bild 3.2: Die Wahrheitstafeln der logischen Operatoren

Shortcurcuit Evaluation

Aus Gründen der Effizienz hat Pascal bei der Auswertung logischer Ausdrücke eine besondere Angewohnheit: Wenn klar ist, daß ein UND-Ausdruck nicht mehr `true` bzw. ein ODER-Ausdruck nicht mehr `false` werden kann, wird dessen Auswertung abgebrochen und `false` bzw. `true` zurückgeliefert.

Das mag im ersten Moment trivial klingen, aber dieses *shortcurcuit evaluation* genannte Verfahren kann unangenehme Konsequenzen haben. Und zwar immer dann, wenn Sie Ergebnisse von Funktionsaufrufen nutzen, um die Ausdrücke zu steuern. Dann kann es nämlich sein, daß bei zwei so miteinander verknüpften Funktionsaufrufen nur noch der erste von beiden ausgeführt wird! Und wenn sich ihr Programm dann aber darauf verläßt, daß auch die andere Funktion ausgeführt wird und eine wichtige Aufgabe erledigt? Tja, dann haben SIE ein Problem! Oder Sie steuern dieses Verhalten mit der Compilerdirektive `{$B}`, mit der dieser Effekt an- oder abgeschaltet werden kann.

Das folgende Beispiel zeigt Ihnen das genauer auf. Und zwar haben wir zwei Funktionen definiert, die immer `true` bzw. `false` zurückgeben. Außerdem sollen Sie noch eine kleine Ausgabe machen, die in diesem Fall angibt, daß die Funktion betreten wurde. Sehen Sie selber, was für Konsequenzen die verkürzte Auswertung hat:

```
 1 program Shorty;
 2
 3 { Beginn der Funktionsdefinitionen }
 4 function truetest : boolean;
 5          begin
 6                    WriteLn('in truetest');
 7                    truetest := true;
 8          end;
 9
10 function falsetest : boolean;
11          begin
12               WriteLn('in falsetest');
13               falsetest := false;
14          end;
15
16 { Beginn des Hauptprogramms }
17 begin
18      { Ordnungsgemäßes Verhalten für UND}
19      WriteLn(truetest and truetest);
20      { Abkürzung für UND }
21      WriteLn(falsetest and truetest);
22
23      { Ordnungsgemäßes Verhalten für ODER }
24      WriteLn(falsetest or falsetest);
25      { Abkürzung für ODER }
26      WriteLn(truetest or falsetest);
27 end.
```

Listing 3.5: Short Circuit Evaluation (🖫 3_5.PAS)

3.2.4 Bitweise Operatoren

Mit Hilfe der bitweisen Operatoren können Sie Ausdrücke bilden, die auf ganz elementarer Ebene mit den einzelnen Bits einer Ganzzahl operieren. Damit ist es möglich, Teile des Hauptspeichers direkt zu

manipulieren, beispielsweise in einem Grafikprogramm oder bei der Steuerung der seriellen Schnittstelle des Computers.

Aber auch für einen einfachen aber trickreichen Verschlüsselungsmechanismus lassen sich solche Operatoren einsetzen. Sie sind direkt aus der Assemblersprache abgeleitet und durch diese direkte Umsetzung auch noch rasend schnell in der Ausführung.

Die möglichen Operatoren

Generell gilt, daß die Operanden der bitweisen Operatoren vom Ganzzahltyp sein müssen und daß der Datentyp des Ergebnisses auch vom Ganzzahltyp ist. Der Grund dafür liegt in ihrer nahen Verwandtschaft zum Assembler.

Hier sind es nicht Strings oder Fließkommazahlen, sondern in erster Linie Bytes (8 Bit) bzw. »Worte« (16 Bit), die manipuliert werden. Und ein Wort entspricht dann dem Pascal-Datentyp `integer`.

Tabelle 3.5: Bitweise Operatoren

Operator	Bedeutung
not	Bitweises NICHT (Bitkomplement)
and	Bitweise UND
or	Bitweises ODER
xor	Bitweises Exklusiv-ODER
shl	Bitweiser Shift nach links
shr	Bitweiser Shift nach rechts

Lassen Sie sich bitte nicht dadurch verwirren, daß die bitweisen und die logischen Operatoren gleiche Namen haben. Aus dem Kontext des Aufrufs, genauer gesagt aus den miteinander verknüpften Datentypen, ermittelt der Compiler, welche Operatoren eingesetzt werden müssen. Manchmal kann das zu unerwarteten Fehlermeldungen führen, wie beispielsweise beim Ausdruck:

```
WriteLn(3 >= 5 and 3 <= 10); { gewünscht:boolscher Vergleich }
```

In diesem Fall meldet der Compiler »Operandentypen passen nicht zum Operator«. Eine einfache Klammerung löst das Problem, und es kommt dann das gewünschte Ergebnis heraus.

```
WriteLn((3 >= 5) and (3 <= 10));
```

Das Komplement

Das Bitkomplement dient dazu, alle Bits in einer Ganzzahl zu »kippen«. Das bedeutet, daß alle 1er mit 0ern ersetzt werden und umgekehrt. Ein Nebeneffekt ist der, daß bei der Ausgabe mittels `WriteLn(not 1)` aus der ursprünglich positiven Zahl 1 ein Wert -2 ausgegeben wird!

Der Grund liegt im Zweierkomplement zur Darstellung negativer Zahlen. Aus `0001` (1) wird nach dem Kippen das Bitmuster `1110` und das bedeutet im Sinne des Zweierkomplements -2.

Das bitweise UND und ODER

Mit dem UND und ODER ist es möglich, auf bestimmte Bitmuster zu testen bzw. ganz gezielt einzelne Bits zu steuern. Die dahinterstehende Logik ist die gleiche wie auch bei den logischen Operatoren:

- Bei UND wird im Ergebnis immer ein Bit gesetzt, wo in beiden Operatoren auch eines gesetzt ist. Damit läßt sich einerseits auf die Wertigkeit eines bestimmten Bits testen, andererseits gezielt ein Bit abschalten.
- Beim ODER wird überall da, wo in einem oder in beiden der Operanden ein Bit gesetzt ist, auch in der Ergebnismenge ein Bit gesetzt. Somit kann man hiermit ein Bit gezielt anschalten.

Im folgenden Beispiel wird eine Zahl mit dem Wert 1 vorbelegt. Das bedeutet, daß bei ihr das erste Bit gesetzt ist. Dann schalten wir das nächste Bit an – mit Hilfe des `or` und dem Bitmuster `...010` (dezimal 2). Das Ergebnisbitmuster `...0011` entspricht dem dezimalen Wert 3. Als letztes schalten wir das erste Bit ab, das neue Bitmuster `...0010` wird durch eine dezimale 2 dargestellt.

```
 1 program Bitmuster;
 2
 3 var
 4      i: integer;
 5
 6 begin
 7      i := 1; { Muster ...0001 }
 8      WriteLn(i);
 9
10      i := i or 2; { mit ..010 verknüpft }
11      WriteLn(i);
12
13      i := i and 2; { mit ..010 verknüpft }
14      WriteLn(i);
15 end.
```

Listing 3.6: Bitmuster steuern (🖫 3_6.PAS)

Das bitweise Exklusiv-ODER

Beim bitweisen Exklusiv-Oder wird im Ergebnis nur da ein Bit gesetzt, wo entweder im einen oder anderen Operanden ein Bit gesetzt war. Für den Fall, daß bei beiden kein oder bei beiden ein Bit gesetzt ist, wird im Ergebnis das Bit »abgeschaltet«. Damit läßt sich ein recht einfacher Verschlüsselungsmechanismus erstellen:

Zuerst nimmt man die zu verschlüsselnde Zahl entgegen und den Schlüsselwert. Beide verknüpft man mittels `xor`. Das Ergebnis ist die verschlüsselte Zahl.

```
Zahl:      15  ...00001111
Schlüssel:  3  ...00000011
Ergebnis:  12  ...00001100
```

Zur Entschlüsselung muß man die verschlüsselte Zahl nur wieder mit dem Schlüssel und `xor` verknüpfen. Daraus entsteht wieder das ursprüngliche Bitmuster:

```
Zahl:      12  ...00001100
Schlüssel:  3  ...00000011
Ergebnis:  15  ...00001111
```

Bitshift-Operatoren

Mit den Bitshift-Operatoren können Sie Bitmuster innerhalb einer Ganzzahl nach links oder nach rechts verschieben. Dabei ist die Zahl der Bitpositionen, um die nach links oder rechts verschoben wird, ein frei wählbarer Wert.

Was passiert mit den Bits, die dabei aus der Zahl »herauskippen«? Nichts, sie sind verloren. Nachgefüllt wird beim Schieben nach links von rechts mit 0, die herausgefallenen Bits werden verworfen.

Wenn Sie einen Bitshift nach rechts vornehmen, kommt es beim »Nachfüllen« darauf an, was für ein Wert an der am meisten links stehenden Bitposition steht:

- Handelt es sich um eine 1, wird mit 1en nachgefüllt,
- Bei einer 0 wird mit 0en nachgefüllt.

Das bedeutet nichts weiter, als daß beim Schieben nach rechts der arithmetische Wert des Vorzeichens der Zahl erhalten bleibt.

Beispiel für die Bitshift-Operatoren

Sie können dieses Verfahren beispielsweise dazu benutzen, um alle Bits einer Zahl nacheinander abzutesten. Das macht man gemeinhin so, daß eine sogenannte Maske angelegt wird, bei der meist das am weitesten rechts stehende Bit gesetzt wird. In einer Schleife wird es dann mit dem zu testenden Bitmuster verglichen. Bei jedem Schleifenschritt wird das Maskenbit um eine Position nach rechts geschoben, wodurch man eine neue Maske erhält, die genau diese Bitposition »befragt«. Und so arbeiten Sie sich schrittweise durch die Überprüfung hindurch.

```
 1 program Bitschiebereien;
 2
 3 var
 4    i, eingabe: integer;
 5    maske: longint; { Wegen Maskengröße diesen Typ }
 6
 7 begin
 8      Write('Ihre Zahl bitte: ');
 9      ReadLn(eingabe);
10
11      maske := $8000; { Muster: 1000000...000 }
12
13      for i := 1 to 16 do
14          begin
15               if eingabe and maske = maske then
16                    Write('I')
17               else
18                    Write('O');
19               maske := maske shr 1;
20          end;
21
22      WriteLn; { abschliessender Zeilenumbruch }
23 end.
```

Listing 3.7: Erzeugen des Bitmusters einer Zahl (🖫 3_7.PAS)

Beim Programm gehen wir bei der Eingabe von Ganzzahlen des Typs `integer` aus. Die Variable für die Maske muß ein erstes Bitmuster tragen, bei dem das erste Bit gesetzt ist. Leider läßt der Compiler dies für die einfache `integer`-Variable nicht zu, weswegen wir eine `longint` nehmen.

In Zeile 11 wird die Startmaske in die Variable eingetragen. Die Schleife zwischen Zeile 13 und 20 schiebt dann das eine Maskenbit unter dem Bitmuster der eingegebenen Zahl durch. Getestet wird durch die Verknüpfung mit `and`, ob für die gegebene Bitposition der Maske auch in der Eingabe ein Bit gesetzt ist. Wenn ja, erfolgt die

Ausgabe von »'1'«, ansonsten von »'0'«. Danach wird das Maskenbit gewissermaßen um eine Position nach rechts verschoben, und der nächste Durchlauf kann beginnen.

3.3 Regeln für die Typumwandlung

Wie wir Ihnen immer wieder mitteilten und Sie bei Ihren Experimenten sicherlich schon feststellen konnten, ist Pascal wirklich recht typstreng, was die Abbildung unterschiedlicher Datentypen aufeinander angeht.

3.3.1 Gleich und gleich ...

Die Grundregel ist die, daß es in Pascal nicht erlaubt ist, unterschiedliche Typen gemischt zu verwenden. Angenommen, Sie haben eine Zeichenkette, in der `'1234'` steht, und nun wollen Sie dazu den numerischen Wert `1` addieren. Es wäre sicherlich eine erste Idee `'1234' + 1` zu schreiben.

Numerisch zu numerisch, Zeichen zu Zeichen

Allerdings handelt es sich eben um eine Zeichenkette und einen numerischen Wert, während die Addition nur für gleiche Typen definiert ist. Also wird der Compiler die Übersetzung mit einer Fehlermeldung abbrechen. Erlaubt sind somit:

```
1 + 1
1.1 + 1.1
1.1 + 1
1 + 1.1
»'1234'« + '1' { ergibt »'12341'« }
```

Verboten sind:

```
»'1234'« + 1
»'1234'« + 1.1
```

Probleme bei Zuweisungen

Ein anderer Fall, wo die Typstrenge zuschlägt, ist die Zuweisung von Werten an eine Variable. Eine Variable kann nur die Inhalte aufnehmen, für die ihr Datentyp vorgesehen ist. So ist es nicht möglich, einer Stringvariablen den Wert `1234` zuzuweisen. Es handelt sich um eine Ganzzahl, keinen String (»`'1234'`«). Damit ist erst einmal eine Umwandlung nötig. Genauso können Sie nicht einer Ganzzahlvariablen einen Fließkommawert zuweisen.

Umgekehrt dagegen ist es schon möglich, einer Fließkommavariablen einen Ganzzahlwert zuzuweisen, der dann genauso wie in der Operation `1.1 + 1` an den Fließkommatyp angepaßt wird.
Der Vorteil dieser Typstrenge ist, daß Sie niemals versehentlich falsche Variablenzuweisungen vornehmen können. Solche Fehler werden nicht erst zur Laufzeit des Programms bemerkt, sondern schon bei der Übersetzung. Damit ist eine Fehlerquelle beim Aufruf von Funktionen beseitigt, die in anderen Programmiersprachen immer wieder zu Verdruß führen kann. Eine Typumwandlung muß dann explizit durch Sie vorgenommen werden.

3.3.2 Mit Gewalt geht (fast) alles

Manchmal muß man also den Compiler überlisten, gewissermaßen »zu seinem Glück zwingen«. Das ist immer dann der Fall, wenn Sie typinkompatible Werte miteinander vergleichen oder gemeinsam weiterverarbeiten müssen.

Erzwungene Typumwandlungen

Dann kommt die Stunde der erzwungenen Typumwandlung, und zwar mit Hilfe von Funktionen. Es stehen einige Funktionen zur Verfügung, die beispielsweise eine Zeichenkette in eine Zahl verwandeln bzw. eine Zahl in eine Zeichenkette. Damit können Sie dann diese Typen ineinander umwandeln.

Wir haben im folgenden eine kleine Liste von Funktionen zusammengestellt, mit denen Sie nützliche Umwandlungen vornehmen können.

Dabei haben wir den Begriff »Umwandlung« allerdings recht weit gefaßt. Denn die Funktion `Ord()` beispielsweise ist nicht direkt eine Umwandlungsfunktion, läßt sich jedoch ganz gut einsetzen, um `true` und `false` bzw. einen Buchstaben in die dazu passenden numerischen Werte umzuwandeln.

Tabelle 3.6: Umwandlungsfunktionen

Funktion	Umwandlung
Str()	Umwandlung numerischer Wert -> String
Val()	Umwandlung String -> numerischer Wert
Char()	Umwandlung Ganzzahl -> ASCII-Zeichen
Ord()	Liefert die Ordinalzahl des Arguments
Int()	Ganzzahliger Anteil einer Fließkommazahl
Ptr()	Konvertierung von Segment und Offset in einen Pointerwert
StrPas()	Konvertiert einen nullterminierten String in einen Pascal-String
StrPCopy()	Konvertiert einen Pascal-String in einen nullterminierten String
Trunc()	Wandelt einen Fließkommawert durch abschneiden der Kommastellen in einen Ganzzahlwert um

Informationsverluste durch Typanpassungen

Angemerkt sei noch, daß die einzelnen Ganzzahltypen alle zueinander kompatibel sind. Allerdings müssen Sie aufpassen, denn bei der Abbildung eines größeren Typs auf einen kleineren kann es zu Informationsverlusten kommen. Gleiches gilt auch für die verschiedenen Fließkommatypen.

3.3.3 C/C++ läßt grüßen!

Eine Erweiterung, die in Borland Pascal 7.0 von C/C++ abgeguckt wurde, ist der sogenannte *Typecast*. In dem Fall wird es möglich, Typen gleicher Bytezahl ineinander umzuwandeln, allerdings auf Ihr eigenes Risiko. Es kann nämlich auch etwas Unsinniges herauskommen.

Vorgehensweise

Die Vorgehensweise ist die, daß Sie, beispielsweise bei einer Zuweisung, vor den Umwandlungskandidaten den neuen Zieltyp angeben, und dann innerhalb eines Klammerpaares den Umwandlungskandidaten selber.

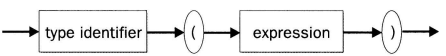


Bild 3.3: Der Typecast

```
erg := Typ(kandidat); { Der Typ von erg entspricht »Typ« }
```

Nötig wurde dieses Gebilde, um die Programmierung von Windowsanwendungen zu erleichtern. Das folgende Beispiel nutzt diese Möglichkeit der Typumwandlung aus, lassen Sie sich vom Ergebnis überraschen!

```
1 program Zeichenketten;
2
3 {$N+}
4
5  var
6    r: single;
7    i: longint;
8
```

```
 9 begin
10      r := 12.3456;
11
12      i:=longint(r);
13
14      WriteLn(i);
15 end.
```

Listing 3.8: Datenmüll durch Typumwandlung (🖫 3_8.PAS)

Die Programmsteuerung 4

4.1 Die Darstellung von Programmen 96

4.2 Verzweigung und Mehrfachauswahl 104

4.3 Schleifen 113

4.4 Sprünge 120

Die bisher gezeigten Programme sind rein sequentielle Abläufe gewesen. Ein Befehl wurde nach dem anderen ausgeführt, und nach Ausführung des letzten war auch das Programm beendet. Solche Abläufe sind für kleinste Aufgaben brauchbar, komplexere Programme lassen sich so nicht realisieren.

Daher gibt es in allen höheren Programmiersprachen Befehle, mit deren Hilfe der Programmablauf beeinflußt werden kann. Das Erstaunliche ist allerdings, daß es dazu gar nicht so vieler Befehle bedarf. Denn im Grunde reicht das Dreigespann

- *Einzelschritt (Sequenz)*
- *Verzweigung*
- *Schleife*

vollständig aus, um alle möglichen Programme zu gestalten. Alles was darüber hinausgeht, ist eigentlich Luxus.

Dieses Kapitel wird Ihnen zeigen, wie Sie die Leistungsfähigkeit von Programmen durch die oben genannten Konstruktionen erweitern können.

Außerdem möchten wir Ihnen einen Tip geben, wie Sie Programme besser planen und dokumentieren können. Denn die drei oben genannten Einheiten lassen sich auch recht gut in grafische Darstellungen von Programmen umwandeln.

4.1 Die Darstellung von Programmen

Eines der besten Mittel, um die Logik eines Programms gut lesbar darzustellen, sind die *Struktogramme (Nassi-Shneiderman* - Diagramme).

Der Vorteil dieser Darstellungsform ist der, daß die Programmlogik damit in einfachen, miteinander kombinierten Grundsymbolen ausgedrückt werden kann. Und zwar weitgehend unabhängig von der später benutzten Programmiersprache! Damit können also COBOL-Spezialisten ohne größere Probleme mit Pascal-Profis über einen Programmentwurf diskutieren.

Andere Verfahren

Es gibt übrigens noch viele andere Verfahren, die Logik eines Programms darzustellen. Unter anderem eine an den Programmiersprachen angelehnte Beschreibungssprache, der sogenannte Pseudocode.

Oder die *Programmablaufpläne* (PAP), die weniger die Struktur, sondern eher den Ablauf des Programmflusses darstellen. Sie sind als Kinder der Assemblerprogrammierung geprägt durch Sprünge und Verzweigungen, ohne ein eigenes Symbol für Schleifen zu bieten.

4.1.1 Die Elemente der Struktogramme

Für alle Standardelemente einer Programmiersprache gibt es grafische Symbole, die sich miteinander kombinieren lassen. Damit nicht jeder seiner eigenen Darstellungsform frönt, wurden die Struktogrammsymbole nach DIN normiert. Wir werden nicht alle Elemente einführen, nur die für uns hier benötigten werden erläutert.

Die Sequenz und der Anweisungsblock

Programme bestehen im einfachsten Fall aus Sequenzen. Das Darstellungsmittel dafür ist ein einfaches Kästchen.

Im Innern steht ein kurzer Text, der beschreibt, was an dieser Stelle zu passieren hat. Es ist gut, wenn Sie keine oder kaum konkrete Anweisungen einer Programmiersprache in solch einen Text einbringen, um das Struktogramm möglichst allgemein und gut übersetzbar zu halten.

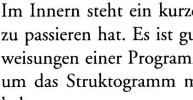


Bild 4.1: Struktogramm einer Sequenz von Befehlen

Größere Programmeinheiten, wie auch das Pascal-Hauptprogramm selber, werden in einem Programmblock dargestellt. Es sind alles Befehle, die dann zum »Programmkörper« gehören. Beispielsweise wird die obige Sequenz innerhalb dieses Blocks eingefügt. Auf die gleiche Art werden Struktogramme auch genutzt, um einzelne Prozedur- oder Funktionsdefinitionen zu gestalten.

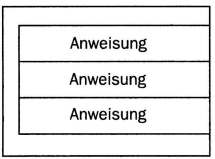


Bild 4.2: Struktogramm eines Programmblocks

Die Verzweigung

Wenn der Ausführungspfad eines Programms sich in Abhängigkeit von einer Bedingung ändert, nennt man das eine Verzweigung. Für den Fall der einfachen »Entweder-Oder- Auswahl« gibt es ebenso ein Symbol, wie für den Fall einer möglichen Auswahl aus mehreren Alternativen.

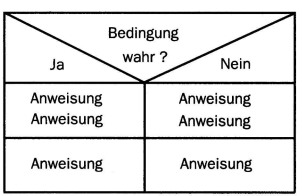


Bild 4.3: Struktogramm der Verzweigung

Im Bild der Verzweigung kommt schön zum Ausdruck, was eigentlich passiert, wenn der Programmablauf auf solch eine Stelle trifft: Zuerst wird die Testbedingung ausgeführt, die hier in dem einführenden Dreieck steht.

Je nach Ergebnis wird dann der linke oder rechte Pfad mit seinen Anweisungen betreten. Dabei ist es durchaus möglich, daß einer der Pfade leer bleibt, weil man für diesen Fall ganz einfach keine besonderen Schritte vornehmen will. Um das zu demonstrieren, wird in das Kästchen, welches den Pfad repräsentiert, einfach ein Kreuz geschrieben.

Genausogut können aber auch mehrere Anweisungen in einem Ausführungsast stehen. Dann werden Sie diese Anweisungen, vielleicht eine Sequenz, in das Kästchen der ausgewählten Alternative eintragen.

Es ist auch möglich, daß innerhalb eines Ausführungsastes auch weitere Verzweigungen, Schleifen oder Mehrfachauswahlen stehen. Diese Schachtelung ist problemlos möglich und kann auch im Struktogramm gut ausgedrückt werden.

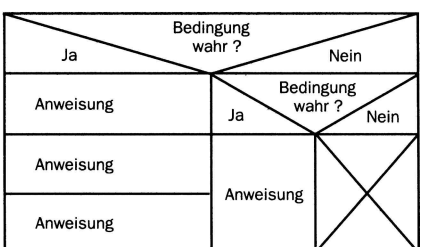


Bild 4.4: Schachtelung von Verzweigungen

Die Mehrfachauswahl

Bei der Mehrfachauswahl wird eine Entscheidung gegen mehrere Alternativen geprüft und dann einer der möglichen Programmpfade eingeschlagen.

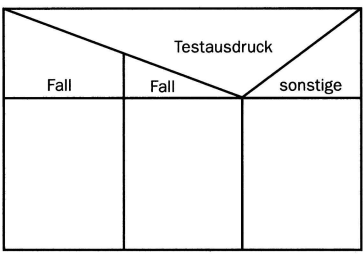


Bild 4.5: Struktogramm der Mehrfachauswahl

In der grafischen Darstellung wird die Mehrfachauswahl mit einem großen Dreieck eingeführt, in dem der zu testende Ausdruck steht. Danach folgen eine ganze Anzahl von alternativen Ausführungspfaden, die jeweils mit einer bestimmten Bedingung verbunden sind.

Sollten Sie auch eine Programmreaktion für die Fälle wünschen, die Sie nicht explizit benannt haben *(Default-Fall)*, dann trägt das Struktogramm der Verzweigung die dazugehörigen Programmbefehle auf der linken Seite im letzten dargestellten Ausführungspfad. Wenn Sie keinen Default-Fall bearbeiten wollen, können Sie diesen Pfad einfach weglassen oder ihn mit einem Kreuz als leer kennzeichnen.

Schleifen

Gesteuerte Wiederholungen von Programmanweisungen sind Schleifen. Eine Bedingung wird immer wieder überprüft, und wenn diese wahr ist, dann wird die Schleife und werden die im *Schleifenkörper* befindlichen Befehle ein weiteres Mal ausgeführt. Wenn die Bedingung nicht mehr zutrifft, wird die Schleife bei nächster Gele-

genheit verlassen, und es wird mit den nach ihr stehenden Anweisungen weitergearbeitet.

Im Aufbau unterscheidet man drei Typen von Schleifen. Es geht dabei um die Frage, an welcher Stelle der Test stattfindet, und darum, ob es sich um eine offene Schleife oder eine Schleife handelt, bei der nach einer genau vorherbestimmten Zahl von Schritten die Abarbeitung beendet wird. Letzteres nennt man auch Zählschleife, und im Grunde ist sie ein Spezialfall der anderen Schleifen.

Wichtig ist die Stelle, an der getestet wird, ob noch ein weiterer Schleifendurchlauf nötig ist. Dabei gibt es zwei Möglichkeiten:

- Man testet, bevor die Befehle des Schleifenkörpers durchlaufen werden. Dann handelt es sich um eine *kopfgesteuerte* Schleife, und im Extremfall kann die Bedingung schon beim ersten Test falsch sein, was zur Folge hat, daß die Anweisungen im Schleifenkörper nie ausgeführt werden.
- Man testet, nachdem die Befehle des Schleifenkörpers ausgeführt wurden. Bei dieser *fußgesteuerten* Schleife ist es so, daß die Anweisungen in ihr immer mindestens einmal ausgeführt werden, auch wenn die Schleifenbedingung von Beginn an nicht wahr gewesen ist.

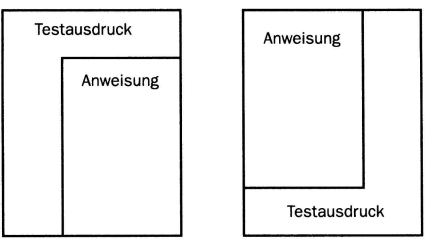


Bild 4.6: Schleifensymbole

Die *Zählschleife* ist dann eine Schleife, bei der schon zum Zeitpunkt der Programmierung exakt bekannt ist, wie viele Durchläufe es geben wird. Meistens wird sie als kopfgesteuerte Schleife realisiert, und in vielen Programmiersprachen gibt es eine eigene Anweisung dafür. Viele Programmierer benutzen das schon vorher erwähnte Symbol für einen Anweisungsblock, um eine solche Schleife im Struktogramm darzustellen.

Funktionen und Prozeduren

Funktionen und *Prozeduren* sind ein Mittel, um große, unübersichtliche Programme in kleinere und leichter beherrschbare Einheiten aufzuteilen. Außerdem haben Funktionen und Prozeduren den Vorteil, daß sie bei einem sauberen Entwurf auch »recycelt« werden können, was den Entwicklungsaufwand bei nachfolgenden Programmen erheblich vermindern kann.

Für die Prozeduren und Funktionen gibt es zwei Symbole im Struktogramm. Für ihre Definition das Blocksymbol, welches Sie vorher schon kennengelernt haben, und das den Prozedurrahmen angibt.

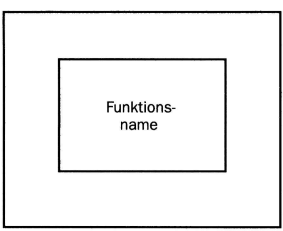


Bild 4.7: Symbol für einen Funktionsaufruf

Daneben existiert noch ein Symbol, mit dem der Aufruf einer solchen Funktion oder Prozedur im Programm selber dargestellt wird. Dieses Symbol setzen Sie also immer dann ein, wenn Sie der Meinung sind, daß ein solcher Aufruf gerechtfertigt ist. In das innere Kästchen

schreiben Sie den Namen der aufgerufenen Funktion oder Prozedur, sowie eventuell beim Aufruf mitzugebende weiterführende Informationen *(Argumente, Parameter)*.

4.1.2 Struktogramm = Programm

Wenn Sie ein Struktogramm anlegen, dann hängen Sie Anweisungen aneinander bzw. fügen sie in verschiedene Verschachtelungsebenen ein. Früher war das ein mühsames Geschäft, alles mußte von Hand gezeichnet werden, Struktogramme als Mittel der Programmentwicklung waren eher »Radierogramme«.

Mittlerweile gibt es verschiedene Programme, bei denen Sie ein Struktogramm interaktiv mit der Maus zeichnen können. Um die Zeichenaufgaben kümmert sich das Programm, entfernte Zweige und Äste werden ohne großen Aufwand beseitigt, neue an beliebigen Stellen eingefügt.

Struktogramm-Editoren

Damit macht es auch Sinn, Struktogramme beim Programmentwurf einzusetzen. Zumal viele dieser Programme aus den Struktogrammen auch Programmgerüste in einer Sprache Ihrer Wahl erzeugen können. Damit ist die halbe Programmierarbeit schon geschrieben. Gleichzeitig sind diese Programme auch in der Lage, aus Quellcodedateien Struktogramme zu erzeugen, die dann mit diesem grafischen Werkzeug dokumentiert und bearbeitet werden können.

Lesen eines Struktogramms

Wie wird ein Struktogramm denn nun gelesen? Nun, Sie lesen es »von oben nach unten«. Meistens ist eine Programm- oder Prozedurbeschreibung in einen Block eingebettet und trägt in der ersten Zeile den Namen und eventuell übergebene Aufrufargumente. Von diesem Punkt aus können Sie dann den einzelnen Elementen schrittweise von oben nach unten folgen, bis Sie zum Ende des umfassenden Blocks gelangt sind.

Auf dem Weg müssen Sie bei Verzweigungen und Auswahlen entscheiden, welcher Weg einzuschlagen ist. Und bei Schleifen müssen Sie die im Schleifenkörper zu findenden Anweisungen entsprechend oft durchlaufen.

4.2 Verzweigung und Mehrfachauswahl

Mit Hilfe von Verzweigungen kann das Programm, während es abläuft, Alternativen auswählen. Verzweigungen können in Pascal fast beliebig ineinander geschachtelt werden und andere Steuerkonstrukte enthalten, geradeso, wie wir es schon bei der Beschreibung der Struktogramme gezeigt haben.

4.2.1 Die Verzweigung mit if

In Pascal leitet das Schlüsselwort `if` eine Verzweigung ein. Es folgt ein Ausdruck, der die zu testende Bedingung darstellt. Dieser Ausdruck muß einen logischen Ausdruckswert, also entweder `true` oder `false`, zurückliefern. Für den Fall, daß die Bedingung wahr ist, wird direkt hinter dem Wort `then` die auszuführende Anweisung geschrieben *(True-Zweig)*.

Ansonsten wird entweder direkt mit dem nachfolgenden Programmteil weitergemacht oder ein besonderer *False-Zweig* durchlaufen.

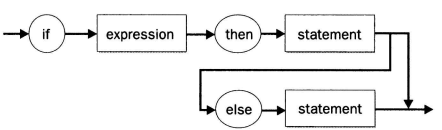


Bild 4.8: Syntaxdiagramm der if-Anweisung

Ein einfacher Test mit if...then

Die einfachste Form der Verzweigung reagiert nach dem Test nur auf den `True`-Fall. Die Reaktion kann aus einem einzigen Ausdruck bestehen oder aus einer Folge von Anweisungen, die dann in einem `begin ... end` -Block geklammert sein müssen.

```
if <Ausdruck> then
    Anweisung;

if <Ausdruck> then
    begin
        Anweisung;
        ...
    end;
```

Beiden Beispielen ist gemeinsam, daß sie auf ein `false` in der Bedingung nicht weiter reagieren.

Damit haben Sie nun zum ersten Mal die Möglichkeit, bei einer Benutzereingabe einen Test auszuführen, ob der Benutzer gültige Werte eingegeben hat. Denken Sie beispielsweise nur einmal an einen Test, ob gerade eine Division durch Null versucht wird:

```
 1 program SichereDivision;
 2
 3 var
 4     erg, wert1, wert2 : real;
 5
 6 begin
 7     Write('Bitte ersten Wert eingeben : ');
 8     ReadLn(wert1);
 9     Write('Bitte Divisor eingeben: ');
10     ReadLn(wert2);
11
```

```
12      if wert2 <> 0 then
13           begin
14             erg := wert1 / wert2;
15             WriteLn(wert1, '/', wert2, '=',  erg:6:3)
16      end;
17 end.
```

Listing 4.1: Beispiel für die Verzweigung (🖫 4_1.PAS)

Mit dem Ausdruck in der Zeile 12 testet das Programm, ob der eingegebene Wert ungleich Null ist, was ja bei der geplanten Division verheerende Folgen haben könnte.

Zwei Wege: if...else

Wenn eine Reaktion auf den `False`-Fall eintreten soll, müssen Sie zusätzlich den `else`-Zweig einführen. In diesem werden dann alle Aktionen stattfinden, die Sie für solch einen Fall vorsehen. Es kann wieder ein einzelner Ausdruck eingesetzt werden, oder ein Anweisungsblock mit `begin...end`.

```
if <Ausdruck> then
    Anweisung
else
    Anweisung;


if <Ausdruck> then
    begin
        Anweisung;
        ...
    end
else
    begin
        Anweisung;
        ...
end;
```

Damit läßt sich auch der Test auf eine fehlerhafte Werteeingabe etwas benutzerfreundlicher gestalten:

```
 1 program SichereDivision2;
 2
 3 var
 4     erg, wert1, wert2 : real;
 5
 6 begin
 7     Write('Bitte ersten Wert eingeben : ');
 8     ReadLn(wert1);
 9     Write('Bitte Divisor eingeben: ');
10     ReadLn(wert2);
11
12     if wert2 <> 0 then
13           begin
14             erg := wert1 / wert2;
15             WriteLn(wert1, '/', wert2, '=',  erg:6:3)
16     end else
17           begin
18             erg := 0;
19             WriteLn('Division durch Null-Fehler!')
20     end;
21 end.
```

Listing 4.2: Beispiel für eine Verzweigung (🖫 4_2.PAS)

Verschachtelte Konstrukte

Natürlich ist es möglich, daß Sie mehrere Verzweigungen ineinander verschachteln. Je nachdem, ob diese Verschachtelung im `True`- oder im `False`-Zweig geschieht, müssen Sie noch etwas berücksichtigen. Und zwar geht es um die Frage, zu welchem `if` ein `else` denn in solch einer Verschachtelung gehört.

```
 1 program Schachtel;
 2
 3 var
 4    i:integer;
 5
 6 begin
 7      { Werteingabe }
 8           Write('Eingabe bitte: ');
 9      ReadLn(i);
10
11      { Variante 1 }
12      if i <= 0 then
13         if i < -10 then
14            WriteLn('kleiner -10')
15         else
16             WriteLn('zwischen -10 und 0')
17      else
18          WriteLn('größer 0');
19
20      { Variante 2 }
21      if i > 0 then {<--- Änderung beachten!!}
22         WriteLn('größer 0')
 23        else
24         if i < -10 then
25            WriteLn('kleiner -10')
26         else
27            WriteLn('zwischen -10 und 0')
28 end.
```

Listing 4.3: Beispiel für verschachtelte Verzweigungen (🖫 **4_3.PAS**)

Im ersten Fall erfolgt für den `True`-Zweig der ersten Abfrage sofort eine weitere Abfrage. Das darauf folgende `else` gehört somit zur zweiten Abfrage und das zweite `else` zur ersten Abfrage.

Die zweite Variante sieht der ersten recht ähnlich, funktioniert aber völlig anders! Nach der ersten Abfrage wird eine Aktion im `True`-

Zweig ausgeführt und damit ist die ganze Sache erledigt. Nur wenn die erste Abfrage erfolglos war, wird deren `else`-Zweig, und damit die dort stehende zweite Abfrage ausgeführt. In diesem Fall gehört also das erste `else` zur ersten Verzweigung und das zweite `else` zur zweiten Verzweigung. Und damit erhalten Sie eben unterschiedliche Ausführungspfade gegenüber dem ersten Beispiel.

4.2.2 Die Mehrfachauswahl

Auch die Mehrfachauswahl hat ihr Äquivalent in einem Programmkonstrukt der Sprache Pascal. Es wird hier, genau wie im Struktogramm, beschrieben, wie eine bestimmte Information gegen andere Werte getestet und in dem Fall, bei dem eine wahre Testbedingung auftritt, der dazugehörige Programmast ausgeführt wird. Und wie es sich gehört, gibt es auch die Möglichkeit, einen »Default-Fall« zu behandeln. Damit läßt sich also das Struktogramm 1:1 in ein Pascal-Konstrukt umwandeln.

Wann Mehrfachauswahl und wann die Verzweigung?

Die Mehrfachauswahl ist immer dann zu bevorzugen, wenn ein Ausdruck gegen mehrere verschiedene mögliche Werte getestet werden soll.

Wenn Sie mehrere Ausdrücke gemeinsam testen müssen, versagt die Mehrfachauswahl. Ebenso haben Sie keine Chance, die Mehrfachauswahl zu benutzen, wenn Strings oder Fließkommazahlen zu bewerten sind. Denn im Zusammenhang mit der `case`-Anweisung dürfen nur die ordinalen Datentypen eingesetzt werden.

Die Syntax der case-Anweisung

Eine Mehrfachauswahl wird eingeleitet mit `case`. Danach folgt ein Ausdruck, dessen Ergebnis ein Ordinaltyp sein muß.

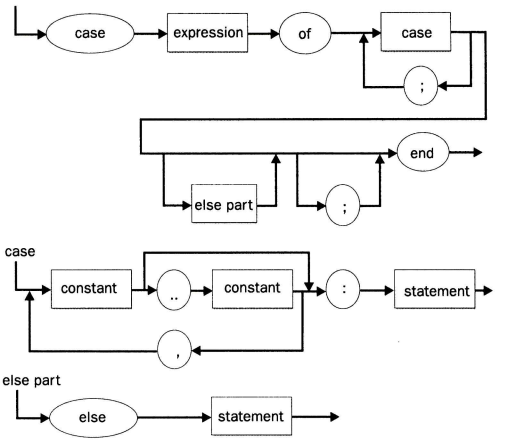


Bild 4.9: Syntax der case-Anweisung

Nach dem Ausdruck folgt ein `of` und dann kann es losgehen: Sie nennen einen konstanten Wert oder einen Wertebereich, der durch zwei Punkte und die jeweiligen Grenzwerte beschrieben wird, oder eine Liste von Konstanten. Bei diesen Konstanten handelt es sich um eben jene Ordinalwerte.

Der Ausdruck im Kopf der `case`-Anweisung wird dann gegen diese Konstanten getestet. Ergibt der Test eine Übereinstimmung, dann wird die nach einem Doppelpunkt folgende Anweisung bzw. der folgende Anweisungsblock ausgeführt.

So können Sie nun einen zu testenden Wert nach dem anderen angeben, immer zusammen mit seinen auszuführenden Anweisungen. Wenn Sie sich dann auch noch um all die Fälle kümmern wollen, die Sie nicht ausdrücklich nennen, dann benutzen Sie das Schlüsselwort `else`. Wenn Sie diese Fälle nicht interessieren, dann lassen Sie es weg.

Für den Fall der eingebundenen Default-Fälle sieht eine Mehrfachauswahl aus wie folgendes Bruchstück:

```
case <Ausdruck> of
    <Wert(e)> : Anweisung;
    <Wert(e)> : Anweisung;
    ...
    else
        Anweisung;
end;
```

Und wenn Sie die Default-Fälle nicht behandeln wollen, sieht die Mehrfachauswahl so aus:

```
case <Ausdruck> of
    <Wert(e)> : Anweisung;
    <Wert(e)> : Anweisung;
    ...
end;
```

Die Abarbeitung einer Mehrfachauswahl

Die Abarbeitung der `case`-Anweisung erfolgt so, daß der Programmfluß hinter dem Fall, der als erster auf die Bedingung paßt, weiterläuft.

Allerdings hat die `case`-Anweisung so ihre Besonderheiten. Und zwar können Sie einen Fall mehrfach angeben! Das bedeutet, daß folgendes Konstrukt erlaubt ist:

```
case wert of
    0..9 :
        WriteLn('0..9');
    5 :
        WriteLn('5');
    5 :
        WriteLn('Fünf');
end;
```

Der Compiler läßt die Sache einfach durchgehen. Wenn jetzt in der Variablen `wert` wirklich einmal die 5 steht, welcher Fall trifft denn dann zu? Die Antwort ist einfach: Der erste Fall, in dem die 5 irgendwie vertreten ist. Und das ist hier auch der erste Fall mit `0...9`. Das bedeutet aber, daß die beiden anderen 5er-Fälle niemals erreicht werden.

Sollten Sie die Reihenfolge einmal umdrehen, was im Rahmen einer Programmveränderung durchaus passieren kann, wird sich das gesamte Programm auf einmal vollkommen anders verhalten, je nachdem, welcher der identischen Fälle nun der oberste ist.

Während Sie den Fehler hier noch ganz gut sehen können, wird es im folgenden Beispiel schon vertrackter. Versuchen Sie einmal, ohne daß Sie den Programmcode ausführen lassen, herauszubekommen, wo denn das Problem liegt:

```
case wert of
    -10...0 :
        WriteLn('Zwischen -10 und 0');
    0...10 :
        WriteLn('Zwischen 0 und 10');
    0 :
        WriteLn('Exakt 0');
    else
        WriteLn('Sonst was');
end
```

Ein Beispiel für die Verwendung der Mehrfachauswahl

Das folgende Programm macht sich die Mehrfachauswahl zunutze, um eine Eingabe zu analysieren. Es stellt fest, ob ein Gleichheitszeichen, arithmetische Operatoren, Ziffern oder sonst etwas eingegeben wurde.

Natürlich hätten wir das auch mit einer mehrfach verschachtelten `if-else`-Verzweigung realisieren können. Allerdings wäre das Ergebnis dann nicht so schön leserlich gewesen. Die von uns gewählte

Formatierung des Programmtextes entspricht unserer Vorstellung von Leserlichkeit und kann von Ihnen jederzeit durch Ihre eigenen Schreibweisen ersetzt werden.

```
 1 program Auswahl;
 2
 3 var
 4    eingabe : char;
 5
 6 begin
 7      Write('Ein Zeichen eingeben: ');
 8      Read(eingabe);
 9
10      case eingabe of
11         '=' :
12             WriteLn('Gleichheitszeichen');
13         '+', '-', '*', '/' :
14             WriteLn('Arithmetischer Operator');
15         '0'..'9' :
16             WriteLn('Ziffer');
17         else
18           begin
19             WriteLn('Was anderes');
20             WriteLn('Damit kann ich nichts anfangen!!!');
21           end;
22      end;
23 end.
```

Listing 4.4: Verwendung der case-Mehrfachauswahl (🖫 4_4.PAS)

4.3 Schleifen

Sie wissen aus der kurzen Beschreibung der Struktogramme, daß es drei Fälle von Schleifen gibt:

- Kopfgesteuerte Schleife
- Fußgesteuerte Schleife
- Zählschleife

Und diese Schleifenformen finden Sie auch in Pascal verwirklicht vor.

4.3.1 Die kopfgesteuerte while-Schleife

Die kopfgesteuerte Schleife ist in Pascal die `while`-Schleife. Hinter dem Schlüsselwort `while` kommt ein Bedingungsausdruck, der immer getestet wird, bevor der Schleifenkörper ausgeführt werden kann.

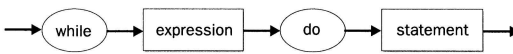


Bild 4.10: Syntaxdiagramm der while-Schleife

Funktionsweise der Schleife

Sollte der Ausdruck `true` sein, werden die Anweisungen des Schleifenkörpers ausgeführt, ansonsten wird mit dem ersten Befehl nach dem Schleifenkörper fortgefahren. Der Schleifenkörper selber kann aus einer einzigen Anweisung oder einem Anweisungsblock bestehen.

Ein Beispiel

Das folgende Beispiel wird eine Multiplikationstabelle erstellen, indem es die `while`-Schleife benutzt.

```
1 program Multitab;
2
3 var
4     x, y, erg : integer;
5
```

```
 6 begin
 7     x := 0; y := 0;
 8
 9     while x < 10 do
10       begin
11            while y < 10 do
12              begin
13                   erg := x * y;
14                   WriteLn(x, ' * ', y, ' = ', erg);
15                   y := y + 1;
16              end;
17              y := 0;
18              x := x + 1;
19       end;
20 end.
```

Listing 4.5: Ineinander verschachtelte Schleifen (🖫 4_5.PAS)

In obigem Beispiel werden zwei `while`-Schleifen ineinander verschachtelt. Man spricht hier von der *äußeren* und der *inneren* Schleife. Die erste benutzt die Variable `x` als *Laufvariable,* die verschiedene Werte von 0 bis 9 durchläuft. Die innere Schleife nutzt dafür die Variable `y`.

Der Programmablauf ist so, daß zuerst `x` auf `0` gesetzt wird und natürlich auch `y`. Die erste Berechnung ist `0 * 0`, Ergebnis: `0`. Dann wird `y` um eins erhöht und nach dem Test, ob die innere Schleife fertig ist, wird `0 * 1` gerechnet, mit dem Ergebnis `0`.

Das geht solange, bis die Laufvariable `y` den Wert `10` erreicht hat. Damit ist aber die Bedingung der inneren Schleife nicht mehr zutreffend und sie wird beendet. Die folgenden Befehle setzen `y` wieder auf `0` zurück und `x` um einen Punkt höher. Da die äußere Schleife damit noch nicht beendet ist, geht das Spiel nun weiter mit `1 * 0`, `1 * 1`, `1* 2` ...

4.3.2 Die fußgesteuerte repeat-Schleife

Die fußgesteuerte Schleife wird immer mindestens einmal ausgeführt. In Pascal ist es die `repeat-until`-Schleife, die dieses Verhalten zeigt.

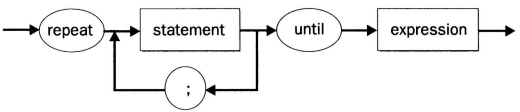


Bild 4.11: Syntaxdiagramm der repeat-Schleife

Funktionsweise der Schleife

In diesem Fall läuft die Programmausführung immer mindestens einmal durch den Schleifenkörper hindurch, bevor der erste Test der Schleifenbedingung erfolgt. Wenn er noch wahr ist, wird der nächste Durchlauf gestartet, ansonsten wird mit der ersten Anweisung nach der Schleife fortgefahren.

Ein Beispiel

Man benutzt diese Form der Schleife beispielsweise bei der Abfrage von Benutzerantworten. Angenommen, eine Berechnung soll solange ausgeführt werden, bis der Benutzer `'Ende'` eingibt. Dann bedeutet das normalerweise, daß diese Berechnung mindestens ein erstes Mal erfolgt, bevor der Benutzer gefragt wird. Trickreich läßt sich das mit einer `while`-Schleife formulieren. Oder Sie benutzen eine `repeat`-Schleife, bei der diese Abfrage glatt und elegant ausgeführt wird:

```
1 program FussSchleife;
2
3 var
4     eingabe : string;
5     erg, i, j : integer;
6
```

```
 7 begin
 8     i := 1; j := 1;
 9     repeat
10           erg := i * j;
11           i := i + 1; j := j + 2;
12           WriteLn('Ergebnis: ', i, '*', j, '=', erg);
13           Write('Weiter mit beliebiger Taste, ');
14           WriteLn('Abbruch mit Ende');
15           ReadLn(eingabe);
16     until eingabe = 'Ende';
17 end;
```

Listing 4.6: Beispiel für eine fußgesteuerte Schleife (🖫 4_6.PAS)

In dem Programm wird die Berechnung mindestens einmal durchgeführt und das Ergebnis ausgegeben. Danach wird dem Benutzer mitgeteilt, wie er das Programm beenden kann und eine Eingabe entgegengenommen wird. Zuletzt erfolgt der Test, wie es denn nun weitergehen soll.

Von »braven« Schleifen und Endlosschleifen

Übrigens haben Sie beim obigen Beispiel das klassische Gegenteil zu einer Zählschleife. Eine Zählschleife ist schon im Moment der Programmierung genau determiniert. Die »offene« Schleife des Beispiels dagegen kann einmal, zweimal, hundertmal oder öfter laufen. Dies ist nicht vorhersagbar und hängt nur von den Launen des Benutzers ab.

Der Extremfall einer offenen Schleife ist eine *Endlosschleife.* Diese bricht nie ab und wird wie der alte Käfer laufen und laufen und laufen...

Manchmal produziert man solch eine Schleife freiwillig:

```
while true do
    { Endlose Ausführung der Anweisungen }
```

Meistens ist sie aber das Ergebnis eines logischen Fehlers, eines Tippfehlers oder eines allgemeinen Programmfehlers. Typischerweise

tritt sie immer dann auf, wenn der Moment, um die Schleife abzubrechen, ganz knapp verpaßt wurde. Das ist meistens bei Fließkommazahlen der Fall und ist unter anderem ein Ergebnis der anfallenden Rundungsfehler. So etwas kann dann wie im folgenden Beispiel zum Desaster führen:

```
while wert <> 3.000 do
    { Hier wird der Wert irgendwie hochgezählt }
```

Der Test `while wert <> 3.000 do` wird in vielen Fällen deshalb danebengehen, weil sich durch die Rundungsfehler ein Übergang von `2.999` auf `3.001` eingeschlichen hat. Die Folge davon ist die, daß der Wert `3.000` niemals erreicht wird, und die Schleife ab jetzt einfach weiterläuft. Es ist eine Endlosschleife entstanden. Im günstigsten Fall läßt sich das Programm abbrechen. Im ungünstigsten Fall müssen Sie den Rechner abschalten!

Deshalb ist es bei Berechnungen mit Fließkommazahlen immer sicherer, wenn Sie den folgenden Test benutzen:

```
while wert < 3.000 do
    { Hier wird der Wert irgendwie hochgezählt }
```

Nun wird die Schleife auch abbrechen, wenn `3.000` nicht ganz exakt getroffen wird.

4.3.3 Die Zählschleife mit for

Durch die `for`-Schleife lassen sich Zählschleifen bequem realisieren. Nach dem Schlüsselwort `for` folgt eine Belegung der vorgesehenen Zählvariable mit einem Startwert. Soll von diesem Wert aus zu einem »höher gelegenen« Zielwert gezählt werden, dann kommt nun ein `to` und der Zielwert. Soll vom Startwert aus heruntergezählt werden, dann steht dort statt dessen ein `downto`.

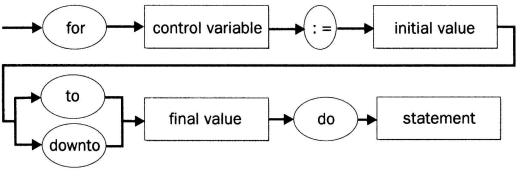


Bild 4.12: Syntaxdiagramm der for-Schleife

Der Schleifenkörper wird mit einem `do` eingeleitet und kann eine einzige Anweisung oder einen Anweisungsblock enthalten.

Funktionsweise und Beispiel

Der Ablauf ist dann so, daß vom Startwert bis zum Endwert inklusive für jeden Zwischenwert die Schleife genau einmal durchlaufen wird.

Das bedeutet, daß beim folgenden Beispiel, die beiden Schleifen 11-mal durchlaufen werden, mit Werten von 0 bis 10 und von 10 bis 0.

```
 1 program ForTest;
 2
 3 var
 4     i : integer;
 5
 6 begin
 7     for i := 0 to 10 do
 8           WriteLn(i);
 9
10     for i := 10 downto 0 do
11           WriteLn(i);
12 end.
```

Listing 4.7: Test der for-Schleife (4_7.PAS)

Für die Veränderung des Zählerwertes müssen Sie nichts unternehmen. Der aktuelle Wert der Variablen steht innerhalb der `for`-Schleife für Sie zur weiteren Verwendung bereit.

Einschränkungen

Leider ist es in Pascal so, daß die Schrittweite der `for`-Schleife festgelegt ist. Es wird immer nur um einen Schritt herauf- oder heruntergezählt. In anderen Programmiersprachen kann man eine beliebige Schrittweite angeben. Eine weitere Einschränkung ist es, daß in Pascal nur die ordinalen Datentypen für die Zähler eingesetzt werden dürfen.

Das bedeutet, daß eine Zählschleife mit Fließkommazahlen als Zählwerte nicht möglich ist. Diese muß mit Hilfe der `while`-Schleife gebastelt werden.

4.4 Sprünge

Mit den Sprüngen ist es so eine Sache. Es gibt Fälle, da kann man nicht anders, um ein gutes und effektives Programm zu schreiben. Aber wenn man sie zu großzügig einsetzt, dokumentiert man lediglich, daß wenig Mühe auf den Programmentwurf verwandt wurde. Für Programme mit zu vielen Sprüngen hat sich der Name »Spaghetticode« eingebürgert.

Seien Sie mit Sprüngen sparsam, dann werden Ihre Programme mit Sicherheit gut strukturiert sein. Und wenn Sie einen Sprung einführen, überlegen Sie lieber noch einmal, ob er nicht vermeidbar ist. Und wenn nicht, dann dokumentieren Sie ihn zumindest so gut es nur geht.

Tabelle 4.1: Eine Liste der Sprungbefehle

Befehl	Einsatz
Break	Beendet eine for-, while- oder repeat-Anweisung.
Continue	Setzt eine for-, while- oder repeat-Anweisung fort.
Exit	Verläßt den momentanen Block mit sofortiger Wirkung.
Halt	Bricht die Ausführung des Programms ab und führt einen Rücksprung zur Betriebssystemebene durch.
RunError	Beendet die Programmausführung.

4.4.1 Das »böse« goto

Mit dem Befehl `goto` können Sie innerhalb eines Programms springen. Das bedeutet, daß je nach Programmsteuerung nicht die nächste Anweisung ausgeführt wird, sondern die Programmausführung an einem ganz anderen Punkt im Programm weitergeht. Die Sprungrichtung liegt dabei nicht fest, sondern Sie können sowohl vor als auch zurück springen.

Sprunglabel

Um solch eine Sprunganweisung einzuleiten, müssen Sie am Sprungziel eine Zielmarkierung, das *Sprunglabel* setzen. Wenn im Programmcode dorthin gegangen werden soll, rufen Sie `goto Name` auf. `Name` ist der Name des Labels.

Damit Sie die Sprungmarkierungen etwas kontrolliert benutzen, müssen Sie die Sprungzeile vorher analog der Variablen extra anmelden.

Beispiel für Spaghetti-Code

Das folgende Beispiel realisiert eine Zählschleife mit Hilfe von `goto`. Wir möchten aber dringend darauf hinweisen, daß wir das nicht für der Weisheit letzten Schluß halten.

```
 1 program GotoTest;
 2
 3 label
 4      Schleifenstart, Schleifenende;
 5
 6 var
 7    i : integer;
 8
 9 begin
10      i := 0;
11      WriteLn('Los geht's);
12
13      Schleifenstart:
14                      if i > 10 then
15                              goto Schleifenende;
16                      WriteLn(i);
17                      i := i + 1;
18                      goto Schleifenstart;
19      Schleifenende:
20
21      WriteLn('Fertig');
22 end.
```

***Listing 4.8: Testfall für goto's* (🖫 4_8.PAS)**

Die Sprungziele unseres Programms werden in Zeile 3 und 4 vereinbart. In Zeile 10 und 11 werden die Zählvariablen initialisiert und eine kurze Meldung ausgegeben. Dann kommt das obere Sprunglabel. Wenn der Programmablauf darauf trifft, hat das keine weiteren Folgen. Die kommen erst mit dem `goto` in Zeile 18, mit dem am Ende des Schleifenkörpers nach dorthin verzweigt wird, um den nächsten Schleifendurchlauf einzuleiten.

Der Test in Zeile 14 überprüft, ob es überhaupt nötig ist, weiter zu machen. Wenn nicht, wird mit dem `goto` nach Zeile 19 verzweigt, und die Schleife ist verlassen, sprich beendet worden. Die Anweisun-

gen des Schleifenkörpers gehen von Zeile 16 bis 17. Die Ausgabe in Zeile 21 ist lediglich ein Abschiedsgruß.

4.4.2 Schleifensteuerung mit break und continue

Was machen Sie, wenn es nötig ist, eine Schleifenausführung vorzeitig abzubrechen? Vielleicht, um ganz aus der Schleife herauszugehen. Oder um zumindest diesen Durchlauf zu überspringen und schon mit dem nächsten Wert weiterzumachen.

Break und continue

Dafür wurden in Pascal die beiden Prozeduren `break` und `continue` geschaffen, die genau das bewirken, was wir gerade beschrieben haben.

- `break`: Bewirkt, daß der von einer `for`-, `while`- oder `repeat`-Anweisung eingeschlossene Block sofort verlassen wird und verläßt die Schleife vorzeitig.
- `continue`: Bewirkt, daß innerhalb einer `for`-, `while`- oder `repeat`-Anweisung der nächste Wiederholungsschritt bearbeitet wird und überspringt so den aktuellen Durchlauf, um mit dem nächsten weiter zu machen.

Beispiele

Im folgenden Beispiel benutzen wir `continue`, um zu verhindern, daß eine Division durch 0 stattfindet. Diese unangenehme Stelle wird einfach übersprungen.

```
1 program TesteContinue;
2
3 var
4     i : integer;
5
```

```
 6 begin
 7     for i := -10 to 10 do
 8           begin
 9                 if i = 0 then
10                                       continue;
11                 WriteLn(1 / i);
12           end;
13 end.
```

***Listing 4.9: Beispiel für den Einsatz von continue* (💾 4_9.PAS)**

Und hier noch eine Verwendung für `break`: In einer Endlosschleife wird der Benutzer aufgefordert, eine Eingabe vorzunehmen. Wenn diese »'Ende'« heißt, dann wird durch `break` die Schleife verlassen.

```
 1 program TesteBreak;
 2
 3 var
 4     eingabe : string;
 5
 6 begin
 7     while true do
 8           begin
 9                 Write('Eingabe : ');
10                 ReadLn(eingabe);
11                 if eingabe = 'Ende' then
12                                       break;
13           end;
14 end.
```

***Listing 4.10: Beispiel für break* (💾 4_10.PAS)**

Auffällig ist, daß bei beiden Beispielen durch eine etwas geschicktere Anordnung der Befehle die Sprünge mit `break` und `continue` hätten vermieden werden können.

4.4.3 Andere Sprünge

`Exit()`, `Halt()` und `RunError()` wollen wir ebenfalls zu den Sprungbefehlen zählen, wenngleich es sich bei `Exit()` genauso wie bei den beiden anderen Befehlen in Wirklichkeit um Prozeduren handelt.

Aufgaben

Mit diesen Prozeduren können Sie den Programmablauf anhalten, und das Programm vorzeitig verlassen. Das macht man oft im Zusammenhang mit Fehlerbehandlungen, wie `RunError()` ja schon andeutet. Immer dann, wenn es brenzlig wird, und ein Programm unbedingt sofort beendet werden muß, oder wenn es einfach keinen einfacheren und eleganteren Weg zur Beendigung gibt, bieten sich diese Prozeduren an.

RunError

`RunError()` veranlaßt dabei die Ausgabe einer Laufzeitfehlermeldung, wobei die Adresse des Punktes angegeben wird, von dem aus `RunError()` aufgerufen wurde.

Halt

Mit `Halt()` wird das Programm verlassen. An die aufrufende Programmumgebung, meist das Betriebssystem, wird ein Fehlercode zurückgegeben. Gerade `Halt()` ist bei der Behandlung von Fehlern im Programm sehr beliebt.

Exit

Und `Exit()` bricht die Programmausführung einfach nur kommentarlos ab.

Beispiel

Das folgende Beispiel benutzt `Halt()`, um beim normalen Programmende einen Code zurückzugeben, der später von anderen Pro-

grammen ausgewertet werden kann. Wir entscheiden uns, alter Programmierertradition gemäß, für den Wert 0.

Sollten Sie aber ein brutales Ende bevorzugen, können Sie dieses veranlassen, und es wird mit `RunError()` ein simulierter Fehlerhalt ausgeführt. Und zwar inklusive Fehlermeldung und allem was dazugehört.

```
 1 program Panik;
 2
 3 var
 4     eingabe : string;
 5
 6 begin
 7     Write('Bei Eingabe von "Panik" kracht's: ');
 8     ReadLn(eingabe);
 9
10     { Rausgeflogen! }
11     if eingabe = 'Panik' then
12           RunError;
13
14     { Ruhiges Ende }
15     Halt(0);
16 end.
```

Listing 4.11: Formen des Programmendes (4_11.PAS)

Öfter mal was Neues 5

5.1 Die Typdefinitionen 128

5.2 Aufzählungen 137

5.3 Unterbereiche 138

5.4 Mengen 140

5.5 Arrays 145

5.6 Records und variante Records 151

Über die bisher besprochenen Datentypen hinaus bietet Pascal Ihnen die Möglichkeit an, eigene Datentypen zu definieren.

Dabei kann es sich um vollkommen neue, nie dagewesene Dinge handeln, oder um Ableitungen und Varianten der schon existierenden Datentypen. Und von diesen selbstdefinierten Datentypen können ebenfalls wieder neue Datentypen abgeleitet werden.

Dieses Kapitel zeigt Ihnen, wie Sie vorgehen müssen, um neue Datentypen zu vereinbaren und einzusetzen. Dabei werden Sie feststellen, daß es in vielen Fällen möglich ist, einen eigenen Datentyp ausdrücklich zu definieren und davon Variablen abzuleiten als auch im Rahmen einer Variablendeklaration die Eigenschaften einer einzigen Variablen direkt einzustellen.

5.1 Die Typdefinitionen

Wir möchten *implizite* und *explizite Typdeklarationen* unterscheiden. Im Fall der expliziten Typdeklaration kommt das Schlüsselwort `type` mit der nachfolgenden Typbeschreibung zum Einsatz.

Implizite Typdeklarationen finden dagegen in dem Moment statt, in dem die Variable deklariert wird. Dabei wird dann der neue Wertebereich, der für die Variable gelten soll, angegeben.

Während mit expliziten Typdefinitionen jeder beliebige Typ angelegt werden kann, sind implizite Typdefinitionen nur für wenige Typen möglich.

5.1.1 Implizite Typdefinitionen

Es ist möglich, daß Sie bei der Deklaration einer Variablen weitere Angaben machen, die gewissermaßen einer Typdefinition gleichkommen.

So ist es nicht unüblich, daß bei der Deklaration einer Zeichenkettenvariablen eine Einschränkung bezüglich ihrer Größe gemacht

wird. Oder für eine Ganzzahlvariable wird angegeben, welche Werte sie annehmen darf.

Daneben existieren auch die Mittel, um Variablen mit ganz neuen Eigenschaften zu erzeugen, beispielsweise *Felder (Arrays)*, *Aufzählungen* oder *Mengen.*

In all diesen Fällen geht Pascal so damit um, als wenn die angelegte Variable von einem eigenen Datentyp ist. Das bedeutet, daß eine Stringvariable der definierten Länge 10 aus der Sicht von Pascal etwas anderes als eine der definierten Länge 100 ist. Und genauso verhält es sich auch mit den anderen Formen impliziter Typdefinitionen.

Der einzige Nachteil ist der, daß Sie, wenn dieser Typ öfter benötigt wird, immer wieder die gesamte Formulierung zur Anlage seiner Variablen angeben müssen. Und es ist nicht möglich, strukturierte Datentypen auf diese Art implizit zu definieren.

Die Form der impliziten Typdefinition

Die Form der impliziten Typdefinition ist recht einfach. Sie ist Bestandteil der normalen Variablendeklaration:

```
var
    name          : Typbeschreibung;
    name2, name3 : Typbeschreibung;
```

Als Typbeschreibung kommen in Frage:

- Aufzählungen: `(element1, element2 ... elementX)`
- Unterbereiche: `startwert...endwert`
- Felder: `array [start...ende] of`
- Mengen: `set of`
- Stringtypen: `string[X]`

Im weiteren Verlauf des Kapitels werden Sie noch genaueres zu diesen Typbeschreibungen erfahren können.

Das Beispiel

Das folgende Beispiel legt einige implizite Typen an. Danach werden die Variablen im Programm mit Werten belegt, und es wird versucht, sie auszugeben. Für die Typen der Aufzählung und der Mengen weiß die Funktion `WriteLn()` nicht, wie die Ausgabe erfolgen soll.

Wenn Sie es trotzdem versuchen, – entfernen Sie ruhig einmal die Kommentare – wird der Compiler eine Fehlermeldung erzeugen. Die restlichen Anweisungen gehen problemlos durch.

Um eine Vorstellung von der Bedeutung dieser impliziten Typen der Variablen zu bekommen, sollten Sie einmal versuchen, ihnen Werte außerhalb ihres Wertebereiches zuzuweisen oder sie auf andere, inkompatible Typen abzubilden.

```
 1 program TypDefTest;
 2
 3 var
 4    colors : (red, green, blue);
 5    bereich : 1..10;
 6    felder : array [1..10] of integer;
 7    worte : string[10];
 8    menge : set of char;
 9
10 begin
11     colors := red;
12     bereich := 10;
13     felder[1] := 4711;
14     worte := 'Hey';
15     menge := [ 'a', 'b', 'c' ];
16
17     { WriteLn(colors); kann so nicht ausgegeben werden }
18     WriteLn(bereich);
19     WriteLn(felder[1]);
20     WriteLn(worte);
21    { WriteLn(menge); kann so nicht ausgegeben werden }
22 end.
```

Listing 5.1: Beispiel für implizite Typdefinitionen (🖫 5_1.PAS)

5.1.2 Explizite Typdefinitionen

Eine explizite Typdefinition beginnt mit dem Schlüsselwort `type`. Danach werden die verschiedenen anzulegenden Typdefinitionen angegeben. Dazu müssen der Typname und die den Typ konstituierenden Bestandteile genannt werden.

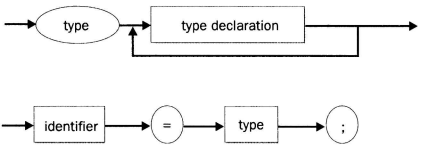


Bild 5.1: Typvereinbarungen

```
type
    Typname = Typbeschreibung;
    Typname = Typbeschreibung;
...
```

Von den so definierten Typen können wiederum eigene Variablen abgeleitet werden, die, je nachdem wie der Typ definiert wurde, unterschiedlich eingesetzt werden.

```
var
    varname: Typname;
    varname: Typname;
...
```

Namenskonventionen

Die Namensgestaltung für neu definierte Typen entspricht der für andere Bezeichner. Somit sind Sie in der Namensvergabe vollkommen frei, sollten aber trotzdem versuchen, sich ein einfaches Schema auszudenken, welches einen selbstdefinierten Datentyp eindeutig identifiziert.

Für dieses Buch werden wir der Konvention folgen, daß die Namen der selbstdefinierten Typen analog zu Funktionsnamen aufgebaut sind. Zusätzlich wird ein `T` vorangestellt, um hervorzuheben, daß es sich um einen selbstdefinierten Typ handelt:

```
type
    TMeinTyp = Typdefinition;
    TDeinTyp = Typdefinition;

var
    var1       : TMeinTyp;
    var2, var3 : TDeinTyp;
```

Ein Beispiel

Abschließend noch einmal eine Wiederholung des Beispiels, welches wir schon bei den impliziten Typdefinitionen eingesetzt haben, diesmal allerdings mit expliziten Typen und daraus abgeleiteten Variablen.

```
 1 program TypeDefProgram;
 2
 3 type
 4    TColors = (red, green, blue);
 5    TBereich = 1..10;
 6    TFelder = array [1..10] of integer;
 7    TWorte = string[10];
 8    TMenge = set of char;
 9
10 var
11    colors : TColors;
12    bereich : TBereich;
13    felder : TFelder;
14    worte : TWorte;
15    menge : TMenge;
16
```

```
17 begin
18     colors := red;
19     bereich := 10;
20     felder[1] := 4711;
21     worte := 'Hey';
22     menge := [ 'a', 'b', 'c' ];
23
24    { WriteLn(colors); kann so nicht ausgegeben werden }
25     WriteLn(bereich);
26     WriteLn(felder[1]);
27     WriteLn(worte);
28    { WriteLn(menge); kann so nicht ausgegeben werden }
29 end.
```

Listing 5.2: Beispiel für explizite Typdefinitionen (5_2.PAS)

Umbenennung existierender Typen

Außerdem läßt sich der Aufbau expliziter Datentypen auch dazu benutzen, schon vorhandene Datentypen umzubenennen, einzuschränken und mit anderen Datentypen zu verbinden.

```
type
    TSize = 1..10;
    TWerte = integer;
    TMesspunkt = TWerte;
    TMessReihe = array [TSize] of TMesspunkt;

var
    w : TWerte;
    m : TMesspunkt;
    wm: TMessReihe;
```

Bitte beachten Sie, wie wir im obigen Codefragment mit der expliziten Typdefinition umgegangen sind.

- Zuerst wurde ein Größentyp `TSize` angelegt, der einen ganzzahligen Bereich von `1` bis `10` umfaßt.

- Als nächstes haben wir einen Typ namens `TWerte` angelegt. Dazu wurde der existierende Datentyp `integer` benutzt. `TWerte` hat nun alle Eigenschaften einer Ganzzahl, und natürlich ist `integer` als Datentyp dadurch nicht verändert worden.
- Der Datentyp `TWerte` erhält gewissermaßen einen zusätzlichen Namen `TMesspunkt`.
- Zuletzt legen wir einen Typ für die Meßreihen (`TMessReihe`) an. Es soll ein Feld von Meßpunkten sein. Die Feldgröße ist über den Größentyp `TSize` definiert. Was wurde also letztendlich angelegt? Ein Feld für `integer`-Werte, welches den Index von 1 bis 10 hat.

Die Möglichkeit, existierende Typen umzubenennen, kann man auch zu folgendem, nicht ganz ernst gemeinten Beispiel einsetzen. Raten sie einmal, was der Compiler Ihnen dazu zu sagen hat!

```
 1 program Pervers;
 2
 3 type
 4    integer = string;
 5
 6 var
 7    i : integer;
 8
 9 begin
10    i := '1';
11    WriteLn(i);
12 end.
```

***Listing 5.3: Unübliche Verwendung der Typdefinition* (🖫 5_3.PAS)**

Der Compiler wird keinen Fehler produzieren. Das Programm wird normal ablaufen. Sie haben in dem Beispiel die Bedeutung des Typnamens `integer` durch die Neudefinition verändert. Ab jetzt sind die Worte `string` und `integer` zueinander synonym!

Das funktioniert deshalb, weil `integer` kein reserviertes Wort ist. Was übrigens bis auf `string` für alle Datentypnamen in Pascal gilt!

Da `string` ein reserviertes Wort ist, wäre die Umdefinition mit diesem Datentyp nicht gelungen.

5.1.3 Typkonstanten

Typkonstanten sind eine Besonderheit von Borland Pascal. Im Standardsprachumfang sind sie nicht enthalten. Sie stellen zwar ein recht nützliches Konstrukt dar, sind aber mit Eigentümlichkeiten behaftet, die die Art, wie sie eingesetzt werden, zumindest nicht intuitiv faßbar macht.

Definition

Typkonstanten sind Variablen, die im `const`-Abschnitt deklariert und initialisiert werden. Sie werden in Programmen wie normale Variablen benutzt.

Lokale und globale Typkonstanten werden jeweils nur zu Programmbeginn initialisiert, nicht bei jedem erneuten Aufruf der Funktion oder Prozedur!

Damit sind Typkonstanten nicht etwa feste symbolische Konstanten, sondern lediglich Variablen, die auf eine recht seltsame Art automatisch mit Werten vorbelegt werden.

Anlage einer Typkonstanten

Die Anlage einer Typkonstanten erfolgt so, daß zuerst der Variablenname angegeben wird, und dann der Typ und der Vorbelegungswert.

```
const
    start : integer = 1000;
    text  : string = 'Aloha, hey';
    tag   : string[2] = 'Mo'
    CR    : char = #13
    tage  : array [1..7] of string[2] =
            ( 'Mo', 'Di', 'Mi', 'Do', 'Fr', 'Sa', 'So');
```

Einsatz

Die so deklarierten und initialisierten Variablen lassen sich im weiteren Programmverlauf ganz normal einsetzen:

```
 1 program TypKonst;
 2
 3 const
 4     tage   : array [1..7] of string[2] =
 5                ( 'Mo', 'Di', 'Mi', 'Do', 'Fr', 'Sa', 'So');
 6
 7 var
 8     i : integer;
 9
10 begin
11     for i := 1 to 7 do
12            WriteLn(tage[i]);
13     for i := 1 to 7 do
14            tage[i] := 'So';
15
16     for i := 1 to 7 do
17            WriteLn(tage[i]);
18 end.
```

Listing 5.4: Einsatz von Typkonstanten (5_4.PAS)

Eine Typkonstante kann allerdings nicht als Ersatz einer echten symbolischen Konstante eingesetzt werden. Damit ist das folgende Codestück dem Compiler eine Fehlermeldung wert:

```
const
    start : integer = 1;
    ende  : integer = 10;

var
    werte : array [start..ende] of integer;
```

5.2 Aufzählungen

Eine *Aufzählung* ist ein benutzerdefinierter Datentyp, bei dem angegeben wird, welche Elemente in diese Aufzählungsmenge hineinkommen. Dadurch, daß es sich um eine Liste von konkreten Werten handelt, ist eine Aufzählung auch ein ordinaler Typ.

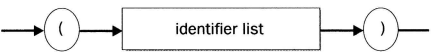


Bild 5.2: Aufzählung

Definition

Eine Aufzählung beginnt immer mit einer öffnenden, runden Klammer. Danach folgen die durch Kommata voneinander getrennten Aufzählungswerte. Beendet wird die Aufzählung mit einer runden, schließenden Klammer.

Die interne Verwaltung erfolgt als einfache Numerierung der einzelnen Werte im Rahmen eines Bytes bei kleinen, oder eines Wortes bei großen Aufzählungsmengen. Auf diesen benutzerdefinierten Datentyp ist `WriteLn()` daher nicht anwendbar!

Einsatz

Wie schon im vorangehenden Abschnitt gezeigt wurde, kann die Aufzählung sowohl implizit als auch explizit eingesetzt werden.

```
1 program Aufzaehlungen;
2
3 type
4   TWoche = ( Mon, Die, Mit, Don, Fre, Sam, Son );
5   TAmpel = ( Rot, Gelb, Gruen );
6
7 var
8   tag        : TWoche;
9   hauptampel : TAmpel;
```

```
10
11 begin
12   { Wertebelegung }
13   tag := Mon;
14   hauptampel := Rot;
15
16   { Auswertung }
17   case tag of
18           WriteLn('Arbeitswoche');
19     Fre, Sam, Son:
20           WriteLn('Wochenende');
21   end;
22
23     case hauptampel of
24           Rot  : begin
25                  WriteLn('Stop');
26                  hauptampel := Gruen;
27                  end;
28           Gelb : begin
29                  WriteLn('Achtung');
30                  hauptampel := Rot;
31                  end;
32           Gruen: begin
33                  WriteLn('Los');
34                  hauptampel := Gelb;
35                  end;
36     end
37 end.
```

Listing 5.5: Beispiele für Aufzählungen (🖫 5_5.PAS)

5.3 Unterbereiche

Eine weitere Variante benutzerdefinierter Ordinaldatentypen sind die *Unterbereiche*. Mit ihnen können Sie einen von den Ordinalzahlen

abgeleiteten Datentyp schaffen, der eine Bereichseinschränkung darstellt.

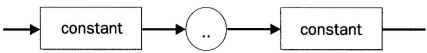


Bild 5.3: Unterbereiche

Definition

Der Aufbau der Unterbereichsdefinition ist denkbar einfach. Sie geben eine untere Grenze, zwei Punkte und die obere Grenze an.

```
type
    TPositiveWerte = 0..32767;
    TNegWerte = -32768..-1;
    TGrossZeichen = 'A'..'Z';
    TKleinZeichen = 'a'..'z';
```

Sie können auch Werte aus Aufzählungsdatentypen für die Vereinbarung von Unterbereichsdatentypen benutzen. Dies geht deshalb, weil beides ordinale Datentypen sind.

Einsatz

Unterbereiche können beispielsweise in der Definition anderer Datentypen eingesetzt werden. Und natürlich auch für die Deklaration von Variablen, die dann aber nur die in den Unterbereichen angegebenen Werte annehmen dürfen.

Wenn Sie beispielsweise einen Datentyp für die Verwaltung von Jahrestagen benötigen, und für diesen Datentyp den Wertebereich auf 1 bis 365 Tage einschränken wollen, müßten Sie das wie folgt vornehmen:

```
...
type
    TMonate = ( Jan, Feb, Mar, Apr, Mai, Jun,
                Jul, Aug, Sep, Okt, Nov, Dez);
    TErstesQuartal = Jan..Mar;
    TZweitesQuartal = Apr..Jun;

    TMonatsTage = 1..31;
    TJahresMonate = 1..12;
    TJahrhundert = 0..99;
    TJahrTage = 1..365;

var
    jahr : TJahrhundert;
    quartal1 : TErstesQuartal;

begin
    jahr := 96;
    quartal1 := Jan;
...
```

Die von diesem Typ definierten Variablen nehmen jetzt nur noch Werte aus den genannten Unterbereichen an. Sollten Sie etwas anderes versuchen, wird der Compiler das nicht zulassen. Allerdings muß man dieses Compilerverhalten auf konstante Werte einschränken, die schon zur Übersetzungszeit angegeben werden. Bei einer Wertezuweisung zur Laufzeit, beispielsweise aus einer Variablen, wird normalerweise keine Überprüfung vorgenommen.

5.4 Mengen

Mit dem *Mengentyp* können Sie verschiedene Objekte ordinalen Typs zusammenfassen. Dazu beschreiben Sie den Datentyp, aus dem die Menge gebildet werden kann. Im späteren Einsatz können bis zu 256 Elemente dieses Datentyps in beliebiger Reihenfolge in die Menge aufgenommen werden.

Definition

Die Definition einer Menge beginnt mit `set of`. Danach folgt eine Angabe des Datentyps, der in der Menge aufgenommen werden kann. Als Datentypen sind alle ordinalen Datentypen erlaubt, also auch Aufzählungen und Unterbereiche. Letztere können sowohl direkt oder als Typname angegeben werden.

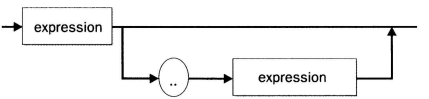


Bild 5.4: Syntax des set-Typs

```
type
    TTage = (Mon, Die, Mit, Don, Fre, Sam, Son);
    TFarben = set of (Rot, Gelb, Gruen, Blau);
    TZiffern = set of 0..9
    TZeichen = set of Char;
    TArbeitstage : set of TTage
```

Einsatz

Die Anwendung erfolgt so, daß zuerst normale Variablen der Mengentypen angelegt werden. Ihre Wertebelegung erfolgt dann mit der Angabe der gerade gewünschten Werte in eckigen Klammerpaaren *(Mengenkonstruktoren)*. Die Elementmenge muß nicht sortiert sein.

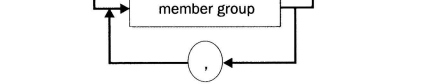


Bild 5.5: Elementaufzählung

```
var
    RGBfarben : TFarben;
    antwort : TZeichen;
    arbeitswoche : TArbeitstage

begin
    RGBFarben := [ Rot, Gelb, Blau ];
    antwort := [ 'j', 'J', 'y', 'Y', 'n', 'N' ];
    arbeitswoche := [ Mon, Die, Mit, Don, Fre ];
```

Die Belegung der Mengenvariablen kann durch explizite Angabe der einzelnen Elemente erfolgen. Allerdings ist es auch möglich, mit Ausdrücken oder mit Bereichsangaben zu operieren. Damit sind Formulierungen wie die folgenden ebenfalls möglich:

```
arbeitswoche := [Die]
arbeitswoche := [Mon..Fre];
RGBFarben := [ Rot, Rot + 1, Rot + 2];
```

Natürlich können die verschiedenen Belegungsarten auch gemischt benutzt werden. Eine *leere Menge* entspricht eckigen Klammern ohne Inhalt `[]`.

Abfrage eines Elements

Die Abfrage, ob ein konkreter Wert in einer Menge enthalten ist, erfolgt mit dem Schlüsselwort `in`. Mit dem Ausdruck *`var`* `in menge` testen Sie, ob das in `var` enthaltene Element in der genannten Menge steht. Die Stelle, an der das Element in der Menge eingeordnet ist, ist dabei nicht weiter bedeutungsvoll.

```
if wert in Menge then ...
while eingabe in ErlaubteZeichen ...
```

Das folgende Beispiel durchläuft eine Schleife solange, bis der Benutzer mit der Eingabe einer passenden Antwort abbricht. Im Gegensatz

zu den bisher dafür benutzten Lösungen wird die Eingabe nicht einzeln abgetestet, sondern über eine Menge ausgewertet:

```
 1 program Mengen;
 2
 3 type
 4     TAntwort = set of char;
 5
 6 var
 7    eingabe : char;
 8    abbruch : TAntwort;
 9
10 begin
11
12    abbruch := ['j', 'J', 'y', 'Y'];
13
14    repeat
15          Write('Abbrechen?');
16          Read(eingabe);
17    until eingabe in abbruch;
18
19 end.
```

Listing 5.6: Beispiel für den Einsatz von Mengen (💾 5_6.PAS)

Operatoren und Funktionen

Mit den Mengen kommen auch noch einige Operatoren hinzu. Sie leiten sich aus den schon bekannten Operatoren ab, die in ihrer Bedeutung auf Mengen erweitert wurden.

Außerdem gibt es noch einige spezielle Funktionen für den Umgang mit Mengen.

Tabelle 5.1: Operatoren und Funktionen für Mengen

Operator/Funktion	Bedeutung
+	Vereinigungsmenge, d.h. die Menge, in der alle Elemente aus der ersten, der zweiten und beiden enthalten sind.
-	Differenz, d.h. die Menge der Elemente, welche in der ersten, nicht jedoch in der zweiten Menge enthalten sind.
=	Identität
<=	Untermenge, d.h. die erste Menge soll in der zweiten enthalten sein.
>=	Obermenge, d.h. die zweite Menge soll in der ersten enthalten sein.
*	Schnittmenge, d.h. die Menge der Elemente, die in beiden Mengen enthalten ist.
in	`true`, wenn das Element in der Menge enthalten ist.
Include()	Fügt in die Menge ein neues Element ein.
Exclude()	Entnimmt der Menge ein Element.

```
 1 program MengenOperatoren;
 2
 3 var
 4     klein, gross, alle : set of char;
 5     ergebnis : set of char;
 6
 7 begin
 8     klein := ['a'..'z'];
 9     gross := ['A'..'Z'];
10     alle  := ['a'..'z','A'..'Z'];
11     ergebnis := [];
12
13     WriteLn(klein <= alle, gross <= alle);
14     WriteLn(alle >= klein, alle >= gross);
```

```
15
16      ergebnis := klein * alle;
17      if ergebnis = klein then
18      if ergebnis = klein then
19              WriteLn('Ergebnis sind die Kleinbuchstaben');
20
21      ergebnis := klein + gross;
22      if ergebnis = alle then
23              WriteLn('Ergebnis sind alle Buchstaben');
24 end.
```

Listing 5.7: Beispiel für die Verwendung der Mengenoperatoren (5_7.PAS)

5.5 Arrays

Manchmal sind viele Variablen gleichen Typs zu verwalten, die auch noch logisch miteinander zusammenhängen. Dann kann es extrem aufwendig werden, wenn mit einzelnen Variablen gearbeitet wird. Dies kann der Fall sein, wenn 100 Meßwerte aufgenommen und verwaltet werden sollen:

- Es werden 100 Einzelvariablen nötig.
- Der Zugriff auf eine einzelne Variable der Gruppe muß über eine komplizierte Mehrfachauswahl realisiert werden.

Um diese Aufgaben zu vereinfachen, wurde der Datentyp der Arrays geschaffen.

5.5.1 Definition eines Arrays

Ein *Array* ist ein Datentyp, der aus einer festen Anzahl von Elementen gleichen Typs besteht. Das gesamte Feld von Einzelvariablen wird über den Arraynamen angesprochen, ein einzelnes Element durch seine Position relativ zum ersten Element. Diese Position nennt man Index.

Deklaration eines Arrays

Arrays können direkt als Arrayvariablen oder über den Zwischenschritt der Arraytypen vereinbart werden. In vielen Fällen ist die direkte Definition die übliche Art der Arrayvereinbarung.

Die Größe eines Arrays muß zum Zeitpunkt der Deklaration vollständig bekannt sein. Es dürfen nur feste, konstante Werte angegeben werden.

Dynamische Arrays müssen Sie mit Mitteln der noch zu beschreibenden dynamischen Speicherverwaltung realisieren.

```
type
    TArray = array [0..10] of integer;
    TMatrix = array [3..4, 7..8] of real;

var
    avar : TArray;
    mvar : TMatrix;
    ia   : array [1..100] of integer;

begin
    avar[1] := 99;
    mvar[4, 7] := 4711;
    ia[9] := 128;
```

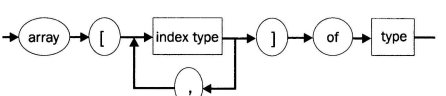


Bild 5.6: Syntax der Arraytypen

Nach dem Schlüsselwort `array` folgt ein Paar eckiger Klammern, zwischen denen Sie die Ausdehnung des Arrays angeben. Die Dimensionierung erfolgt durch die Angabe der obersten und untersten

Indexgrenzen des Arrays. Als Dimensionsangabe dürfen nur Ordinalwerte eingesetzt werden.

Mehrdimensionale Arrays

Wenn es sich um ein eindimensionales Array handelt, dann muß nur eine Größe angegeben werden. Im Fall eines mehrdimensionalen Arrays *(Matrix)* müssen die einzelnen Dimensionen durch Kommata getrennt innerhalb der eckigen Klammern angegeben werden. Es ist dabei zu beachten, daß Sie immer eine Ausdehnung der Art `von..bis` eintragen. Damit ist die Größe und der Verlauf des Indexes genau definiert.

Erlaubte Datentypen

Nach der Angabe der Arrayausdehnungen müssen Sie noch angeben, von welchem Typ die einzelnen Datenelemente sein sollen. Es ist jeder beliebige Typ erlaubt, also auch wieder selbstdefinierte Arraytypen. Beachten Sie bitte, daß zwischen den beiden folgenden Beispielen trotz der ungleichen Definition in der späteren Handhabung keine Unterschiede bestehen!

```
type
    { Ein Array der Größe 10 von Arrays der Größe 10 }
    IArray = array [1..10] of integer;
    DoppelArray = array [1..10] of IArray;
    { Eine Matrix der Größe 10 x 10 }
    Matrix = array [1..10, 1..10] of integer;
...
var
    aa : DoppelArray;
    bb : Matrix;
...
    aa[1,1] = 99;
    bb[1,1] = 99;
```

5.5.2 Zugriff auf Arrayelemente

Die Größe des Arrays ist mit der Angabe seiner Indexgrenzen vollständig bestimmt. Mit dem Arrayoperator und dem Index ist es möglich, auf einzelne Elemente des Arrays zuzugreifen.

Ein Zugriff auf das Array als Ganzes, gewissermaßen eine alle Elemente betreffende Aktion, ist nicht vorgesehen, und führt zu Fehlermeldungen. Trotzdem ist es erlaubt, daß die Werte zweier typidentischer Arrays mit einer einzigen Zuweisung aneinander zugewiesen werden:

```
 1 program ArrayZuweisung;
 2
 3 var
 4     a, b : array [1..10] of string;
 5
 6 begin
 7     a[1] := 'Hey';
 8     b := a;
 9
10     WriteLn(a[1], b[1]);
11 end.
```

Listing 5.8: Beispiel für Arrayzuweisungen (5_8.PAS)

Laufwerte des Index

Der Index des Arrays läuft immer inklusive der beiden Randelemente. Damit ergeben sich die folgenden Konsequenzen für die möglichen Indexwerte und den Platzverbrauch:

- Dimension: 0...10 = 11 Elemente
- Indexlauf: 0,1,2...9,10

- Dimension: 1...10 = 10 Elemente
- Indexlauf: 1,2...9, 10

Zugriff und Zugriffsfehler

Der Zugriff auf ein einzelnes Arrayelement erfolgt durch die Angabe seines Indexes, um so seine Lage in dem Array oder der Matrix zu identifizieren. Das gilt sowohl für direkt als Arrays angelegte Variablen als auch für Variablen, die von Arraytypen abgeleitet wurden.

Ein Zugriff oberhalb oder unterhalb der Arraygrenzen führt in der Standardeinstellung des Compilers zu einem Laufzeitfehler. Wenn Sie ein Array mit beispielsweise den Grenzen `5..10` definiert haben, wird es bei einem versuchten Zugriff auf die Elementposition 4 oder 11 zu Fehlermeldungen kommen.

```
type
    IArray = array [1..10] of integer;
...
var
    ia = IArray;
    ra = array [1..10] of real;
...
    ia[1] = 99;
    ra[1] = 9.9;
```

Allerdings läßt sich diese Bereichsüberprüfung auch via Compilerschalter abschalten. Sollte es dann zu einem Zugriffsfehler kommen, ist der weitere Programmablauf nicht vorherzusagen. Von gar keiner Reaktion über Datenverlust bis zum Systemabsturz ist alles möglich!

Ein Beispiel

Das folgende Beispiel benutzt ein Array von Strings, um die Eingabe der Benutzer entgegenzunehmen, und diese dann abzuspeichern. Bitte beachten Sie, daß für die Anlage der Datei keinerlei Sicherheitsvorkehrungen getroffen wurden.

```
1 program DateiAnlage;
2
3 const
4     MAXTEXT = 100;
5
```

```
 6 type
 7     TText = array [1..MAXTEXT] of string;
 8
 9 var
10     textzeilen : Ttext;
11     eingabe : string;
12     znr  : integer;
13     i     : integer;
14     dname : string;
15     datei : Text;
16
17 begin
18     znr := 1;
19     eingabe := '';
20
21     while (znr <= MAXTEXT) and (eingabe <> '.') do begin
22          Write(znr:5, '> ');
23          ReadLn(eingabe);
24          if eingabe <> '.' then begin
25                textzeilen[znr] := eingabe;
26                znr := znr + 1;
27          end;
28     end;
29
30     Write('Dateiname zum Speichern: ');
31     ReadLn(dname);
32
33     Assign(datei, dname);
34     ReWrite(datei);
35
36     for i := 1 to znr do
37         WriteLn(datei, textzeilen[i]);
38
39     Close(datei);
40 end.
```

Listing 5.9: Anlage eines Textes (💾 5_9.PAS)

Wir haben für die maximale Arraygröße eine Konstante angelegt, was uns spätere Änderungen leichter macht. Das Array ist vom Typ `string`.

Im Programmverlauf gibt der Benutzer seine Zeilen ein, und ein einsamer Punkt auf der Eingabezeile beendet den Vorgang. Der Punkt wird übrigens nicht mit gespeichert. Die Eingabezeilen werden im Array gesammelt. Beachten Sie die Abfrage, mit der wir verhindern, daß zu viele Zeilen eingegeben werden.

Der Dateiname wird als nächstes abgefragt, die Datei angelegt und alle eingelesenen Zeilen in die Datei übertragen. Das ist fast schon die Funktionalität eines kleinen Editors, oder?

5.6 Records und variante Records

Die bisher beschriebenen benutzerdefinierten Datentypen zeichnen sich durch eine gewisse Uniformität aus. Für viele Aufgaben mag das ja ausreichen. Manchmal jedoch ist die zu lösende Aufgabe zu komplex, um sie mit solch einfachen Mitteln zu lösen.

Ein kleineres Beispiel dafür ist die Verwaltung von Adreßdaten. Diese Daten verlangen zumindest nach Zeichenketten für Namen, Vornamen, Straße, PLZ, Ort und Telefonnummer. Sechs Einzelkomponenten machen eine komplette Adresse aus.

Angenommen, Sie erledigen das mit sechs einzelnen Variablen. Die Aufgabe ist für einen oder zwei Adressen so noch gut handhabbar. Aber wenn hundert Adressen verwaltet werden sollen, bleiben Ihnen nur sechs Arrays.

Logisch zusammengehörige Daten in einem Datentyp

Irgendwie ist es verrückt, wenn diese sechs Daten, die eine logische Einheit bilden, in sechs unterschiedlichen und voneinander getrennten Datentypen verwaltet werden.

Eine viel natürlichere Lösung ist es, wenn man einen neuen Datentyp schafft, der sich aus den sechs Einzelkomponenten zusammensetzt,

und man dann nur noch ein Array mit hundert Einträgen zu verwalten hat.

Solch eine Möglichkeit, strukturierte Datentypen zu gestalten, gibt es wirklich in Pascal: Die Struktur, auch als *Record* bezeichnet.

5.6.1 Records

Pascal bietet mit dem Datentyp der Strukturen eine Möglichkeit an, mit der ein Datentyp geschaffen werden kann, der aus einer genau bestimmten Anzahl von einzelnen Unterelementen aufgebaut ist.

Aufbau

Die Strukturelemente können ihrerseits ebenfalls wieder Strukturen sein. Somit wird es möglich, auch sehr komplexe Datenmodelle in einfach handhabbare Einheiten zusammenzufassen.

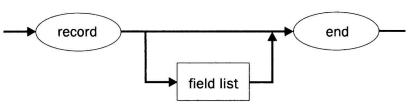


Bild 5.7: Syntaxdiagramm

Die Strukturdefinition wird nach einem `type` mit dem Namen des neuen Datentyps eingeleitet. Die Strukturdefinition selber beginnt mit dem Schlüsselwort `record` und ist mit einem `end` abgeschlossen.

Dazwischen können die einzelnen *Strukturelemente* definiert werden. Der Stil dieser Definitionen ist der gleiche wie bei normalen Variablenvereinbarungen auch.

Um von solch einem selbstdefinierten Datentyp eine Variable oder mehr anzulegen, gehen Sie genauso vor, wie bei den vordefinierten Datentypen. Und auch sonst können Sie alles, was mit den vordefi-

nierten Datentypen erlaubt ist, auch mit Ihren selbstdefinierten machen:

```
type
    TTage = (Mon, Die, Mit, Don, Fre, Sam, Son);

    TDatum = record
        tag : TTage;
        monat : 1..12;
        jahr : 1900..2100;
    end;

    TJahre = array [1..365] of TDatum;

var
    heute : TDatum;
    tage : array [1..365] of TDatum;
    jahre : array [1..100] of TJahre
```

Somit können wir nun auch die Aufgabe mit dem Adreßdatentyp bequem und elegant lösen:

```
type
    TAdresse = record
        name : string[30];
        vname : string[10];
        strasse : string[30];
        plz : string[5];
        ort : string[30];
        tel : string[20];
    end;

var
    adressen : array [1..100] of TAdresse;
```

Bleibt noch die Frage zu klären, wieviel Platz solch eine Struktur verbraucht. Die Definition ist gewissermaßen umsonst. Sie stellt erst einmal nur die Beschreibung für den Fall dar, daß eine Variable dieses Typs vereinbart werden soll.

Erst bei der konkreten Deklaration der Variablen wird Platz verbraucht. Grob geschätzt kann man sagen, daß genau soviel Platz verbraucht wird, wie die Typen der Einzelelemente anfordern. Technisch gesehen kann es sogar sein, daß aus Gründen der Optimierung intern etwas mehr Platz benötigt wird, was Sie aber im allgemeinen nicht belasten sollte.

Damit verbraucht der Adreßdatentyp ca. 125 Byte pro Datensatz, das Array mithin 12.500 Byte. Ganz schön viel, nicht wahr?

Einfacher Zugriff

Der Zugriff auf Strukturen erfolgt im Prinzip genauso wie bei normalen Variablen. Sie müssen lediglich den Strukturnamen angeben, gefolgt von einem Punkt und dem nachfolgenden Elementnamen:

```
Struktur.Element := ...
```

Stellen Sie sich nun vor, Sie würden eine Adreßvariable aus dem oben definierten Array mit Werten füllen:

```
adressen[i].name := 'Watz';
adressen[i].vname := 'Willi';
adressen[i].strasse := 'Watzplatz 4';
adressen[i].plz := '12345';
adressen[i].ort := 'Watzingen';
adressen[i].tel := '0815-4711';
```

Sollte Ihnen diese Vorgehensweise nicht sonderlich bequem erscheinen, haben Sie recht. Und deshalb wurde auch das Schlüsselwort `with` eingeführt, welches das ganze Verfahren etwas benutzerfreundlicher gestaltet.

```
with adressen[i] do begin
    name := 'Watz';
    vname := 'Willi';
    strasse := 'Watzplatz 4';
    plz := '12345';
    ort := 'Watzingen';
    tel := '0815-4711';
end;
```

Strukturen in Strukturen

Nachfolgend zeigen wir Ihnen noch einige Beispiel für verschiedene Strukturen und die Zugriffe auf ihre Elemente. Weitere Beispiele können Sie dann dem Kapitel über moderne Programmiertechniken entnehmen.

Strukturen in Strukturen sind solch eine interessante Form der Datentypen:

```
type
    TS1 = record
        i : integer;
    end;

    TS2 = record
        j : TS1;
        k : integer;
    end;

var
    l : TS2;

    ...
    l.j.i = 99;
    l.k = 100;
```

TS1 besteht – aus praktischen Gründen – nur aus einer Ganzzahlvariablen. Und TS2 aus einer Variablen des soeben definierten neuen Typs und der Variablen k vom Typ integer.

Nachdem wir die Variable l vom Typ TS2 vereinbart haben, erreichen wir das Innere von TS1 über den Weg l.j.i und das Innere von TS2 über den Weg l.k.

Arrays über Arrays

Wie Sie sich sicher denken können, ist es auch möglich, Arrays in Strukturen zu definieren. Es können auch verschachtelte Arrays von Strukturen sein. Und davon kann man natürlich wieder Arraydatentypen definieren, von denen Arrayvariablen deklariert werden ...

```
type
    TS = record
        a : array [1..10] of integer;
    end;

var
    tsa : array a[1..10] of TS;

begin
    ...
    tsa[1].a[1] := 99;
```

Die Struktur enthält ein Array von Ganzzahlen. Wir vereinbaren unsererseits ein Array von dieser Struktur. Der Zugriff erfolgt dann sowohl über die Struktur.Element-Kette, als auch unter Berücksichtigung der benötigten Indizes.

5.6.2 Variante Records

Eine spezielle Form der strukturierten Datentypen sind die *varianten Records*. Mit ihnen wird es möglich, mehrere verschiedene Strukturen in einer Definition zu halten. Je nach *Steuerwert* sucht der Compiler dann die im Moment passende Strukturbeschreibung heraus.

Natürlich kann eine Variable von diesem Datentyp zur Zeit immer nur nach einer ganz genau definierten Struktur aufgebaut sein. Aber je nach Steuerwert kann beim nächsten Aufruf ein anderes Strukturmodell eingesetzt werden. Der gesamte Speicherplatzverbrauch richtet sich nach der Beschreibung der größten Struktur.

Das ist ein seltsames Konzept? Richtig, aber was glauben Sie, welch schmutzige Programmiertricks man damit realisieren kann!

Aufbau

Die Definition eines varianten Record erfolgt im Prinzip genauso wie bei einer normalen Struktur.

Allerdings kommt nun noch ein `case`-Teil hinzu, welcher weitere Strukturdefinitionen enthält. Während des Programmlaufs wird dann aus dieser Ansammlung von Strukturen – in Abhängigkeit vom Steuerwert – die gewünschte herausgesucht.

```
type
    TVarRec = record
        element1 : integer;
        element2 : real;

        case auswahl : integer of
              1 : (celement : char);
              2 : (ielement : integer);
              3 : (relement : real);
              4 : (selement : string);
    end;

var
    tvr : TVarRec;
```

Die Namensvergabe und auch der Typ im `case`-Teil sind vollkommen frei wählbar. Meistens wird man sich wohl auf `boolean`-Werte beschränken, aber, wie Sie sehen können, ist es möglich, eine ganze Liste von Alternativen anzugeben.

Die runden Klammern müssen übrigens sein. Innerhalb des Klammerpaares können wieder beliebig viele Unterelemente definiert sein. Diese sind nur im Zugriff, wenn der passende Steuerwert gesetzt ist.

Außerdem darf ein Elementname grundsätzlich immer nur einmal vorkommen, auch wenn er in unterschiedlichen Strukturauswahlen liegt!

Zugriff

Der Zugriff erfolgt nun in einer etwas komplizierteren Art, da ja auch immer das Steuerelement mit berücksichtigt werden muß.

Für das obige Beispiel müssen wir zwei Fälle des Zugriffs unterscheiden. Da sind die normalen Elementvariablen `element1` und `element2`. Bei ihnen gibt es keine Steuerung, sie sind immer zugreifbar.

```
tvr.element1 := 99;
tvr.element2 := 99.99;
```

Um die varianten Teile zu erreichen, muß zuerst die Steuervariable richtig eingestellt werden. Angenommen, wir wollten einen String speichern, wären zwei Schritte nötig:

```
tvr.auswahl := 4
tvr.selement := 'Aloha, hey';
```

Nun bleibt die Variable `tvr` solange auf der Auswahl für strings, bis der Auswahlwert umgesetzt wird. Setzen wir ihn auf 2, wird im Beispiel nicht mehr `selement`, sondern `ielement` zugänglich sein. Allerdings war da ja noch etwas aus dem vorherigen Schritt drin, oder?

Um Ihnen diese Effekte einmal live zu zeigen, haben wir das folgende Programm geschrieben. Es belegt `tvr` in der Stellung 1 mit dem »`'X'`«. Und dann fragen wir einfach einmal ab, was denn die anderen Schalterstellungen daraus so machen. Lassen Sie sich überraschen:

```
1 program Effekt;
2
3 type
4     TVarRec = record
5           element1 : integer;
6           element2 : real;
7
8           case auswahl : integer of
9                 1 : (celement : char);
10                2 : (ielement : integer);
11                3 : (relement : real);
12                4 : (selement : string);
13    end;
14
15 var
16    tvr : TVarRec;
17
18 begin
19     tvr.element1 := 99;
20     tvr.element2 := 99.99;
21
22     WriteLn(tvr.element1, ' ', tvr.element2);
23
24     tvr.auswahl := 1;
25     tvr.celement := 'X';
26     WriteLn(tvr.element1, ' ', tvr.element2, ' ',
27             tvr.celement);
28
29     tvr.auswahl := 2;
30     WriteLn(tvr.element1, ' ', tvr.element2, ' ',
31             tvr.ielement);
32
33     tvr.auswahl := 3;
34     WriteLn(tvr.element1, ' ', tvr.element2, ' ',
35             tvr.relement);
36
37     tvr.auswahl := 4;
```

```
38      WriteLn(tvr.element1, ' ', tvr.element2, ' ',
39              tvr.selement);
40 end.
```

Listing 5.10: Interessante Effekte mit varianten Records (5_10.PAS)

Dateizugriffe 6

6.1 Einfache Grundoperationen mit Dateien 162

6.2 MS-DOS läßt grüßen 179

In den vorangehenden Kapiteln haben Sie genug Erfahrungen sammeln können, um leistungsfähige Programme zu schreiben. Aber so schön die Programme auch sein mögen, sie haben doch einen gravierenden Fehler: Immer wenn sie beendet werden, ist alles, was bisher geschaffen wurde, weg. Es sei denn, Sie hätten alles aufgeschrieben. Von einem bequemen Datenaustausch mit anderen Programmbenutzern kann da schon gar nicht die Rede sein.

Um dem Dilemma abzuhelfen, benötigen Sie Möglichkeiten des *Dateizugriffs*. Dadurch können Ergebnisse oder Zwischenresultate auf der Festplatte oder auf einer Diskette ausgelagert und beim nächsten Programmstart oder bei passender Gelegenheit im aktuellen Programmlauf wieder geladen werden. Und so wird auch die Weitergabe an andere Personen möglich.

Pascal bietet einige vordefinierte Datentypen und Funktionen, um Ihnen den Dateizugriff so einfach wie möglich zu machen. Von genau diesen Zugriffsverfahren handelt dieses Kapitel, an dessen Ende wir uns außerdem noch mit einigen Funktionen zum Zugriff auf das MS-DOS-Dateisystem befassen.

6.1 Einfache Grundoperationen mit Dateien

Die Behandlung von Dateien ist in Pascal recht gut vorbereitet worden. Sie müssen sich so gut wie gar nicht um die dahinter stehenden physikalischen und organisatorischen Probleme kümmern.

6.1.1 File of ...

Das geht sogar soweit, daß sich eine Datei in einem Pascal-Programm genauso verhält wie eine normale Variable. Lediglich die verschiedenen Operatoren sind auf die *Dateivariable* nicht anwendbar.

Dafür gibt es aber ein großes Repertoire an Funktionen, mit denen Sie sich an eine Datei ankoppeln und sie lesen, schreiben, verändern und löschen können.

6.1.2 Textdateien

Beginnen wir mit dem Zugriff auf *Textdateien.* Textdateien bestehen aus ASCII-Zeichen. Zusätzlich ist ein Text aber auch noch in Absätze, Sätze und Worte organisiert. Daher kann es etwas kompliziert werden, wenn man den Zugriff darauf selber verwalten will.

Datentyp Text

So ist denn der Typ `Text` eine echte Erleichterung. Wenn Sie mit ihm eine *Dateivariable* vereinbart und an eine Textdatei angebunden haben, brauchen Sie sich praktisch um nichts mehr zu kümmern. Sie können sogar die liebgewonnenen Befehle `ReadLn()` zum Lesen und `WriteLn()` zum Schreiben benutzen. Einfacher geht es nun wirklich nicht mehr.

Als Trennzeichen zwischen den einzelnen Datensätzen erkennt Pascal automatisch den Zeilenumbruch und liest nun bei jedem `ReadLn()` aus der Datei die Zeichen bis zum Satzende. Der Text wird dann in eine Programmvariable übertragen und kann so weiter verarbeitet werden.

Beispiel

Genau so einfach ist das folgende Beispiel aufgebaut, mit dem eine Textdatei gelesen und auf dem Bildschirm ausgegeben wird.

```
 1 program TextViewer;
 2
 3 var
 4     datei: Text;
 5     zeile: string;
 6
 7 begin
 8     { Öffnen einer Datei, hier mit vorgegebenen Namen! }
 9     Assign(datei, 'C:\AUTOEXEC.BAT');
10     Reset(datei);
11
```

```
12      { Lesen bis zum Dateiende = End Of File, EOF }
13      while not Eof(datei) do
14              begin
15                      ReadLn(datei, zeile);
16                      WriteLn(zeile);
17              end;
18
19      { Datei schliessen }
20      Close(datei);
21 end.
```

Listing 6.1: Beispiel für das Lesen einer Textdatei (6_1.PAS)

Die Variable `datei` ist im obigen Beispiel vom Typ `Text`, außerdem wurde noch eine Variable für einzelne Zeilen vereinbart. Die Datei wird geöffnet – wir benutzen hier eine mit Sicherheit existierende Datei, damit Sie beim ersten Test nicht zu viele Probleme haben.

Durch `Assign()` wird erst einmal die Verbindung zwischen der Dateivariablen und dem Namen hergestellt. Erst der Aufruf von `Reset()` öffnet die Datei wirklich – wenn sie existiert! Wenn nicht, gibt es hier eine vom Pascalsystem generierte Fehlermeldung. Wir zeigen Ihnen später noch, wie Sie diesen Fehler selbst abfangen und auswerten können.

Dann wird in einer Schleife immer wieder getestet, ob das Dateiende schon erreicht wurde (`Eof(datei)`). Wenn nicht, wird der nächste Satz mit `ReadLn()` ausgelesen und mittels `WriteLn()` auf dem Bildschirm ausgegeben.

Ist die Schleife beendet, schließen wir die Datei mit `Close()` wieder. Das wäre in diesem Fall nicht nötig gewesen, da das Programmende automatisch dafür gesorgt hätte. Allerdings sollten Sie es sich zur guten Gewohnheit machen, eine nicht mehr benötigte Datei sofort selber zu schließen.

Allgemeine Schritte beim Bearbeiten einer Datei

An diesem Programm sehen Sie alles, was auch auf andere Dateien bearbeitende Programme zutrifft. Die regelmäßig auszuführenden Schritte heißen:

- Passenden Dateityp bestimmen
- Dateivariable mit dem Dateityp verbinden (`Assign()`)
- Datei öffnen und den Anfang festlegen (`Reset()`)
- Bearbeiten
- Datei wieder abkoppeln, also schließen (`Close()`)

6.1.3 Typisierte Dateien

Textdateien zeichnen sich dadurch aus, daß in ihnen Sätze unterschiedlicher Länge stehen können. Der eine Satz hat 5 Worte, der andere ist von kantscher Länge. Durch den Typ `Text` weiß Pascal damit umzugehen. Aufwendig ist das allemal.

Dann gibt es aber noch die Dateien, bei denen jeder Satz die gleiche Länge hat, wie es bei Datenbanken oft der Fall ist. Oder bei einer Textdatei, deren Sätze immer exakt N Zeichen lang sind. In dem Fall greift man auf eine andere Art der Deklaration von Dateivariablen zu: *typisierte Dateivariablen.*

Deklaration einer typisierten Dateivariablen

Hier beginnt die Variablendeklaration mit den Worten `File of`, und dahinter geben Sie den Typ an, den die Sätze Ihrer Datei haben sollen.

```
var
    datei: File of Typ;
```

Welchen Vorteil haben Sie davon? Sie müssen sich vorstellen, daß sich Pascal intern merkt, welche Bytes als letzte aus der Datei gelesen bzw. geschrieben wurden. Es handelt sich dabei um den sogenannten *Schreib-/Lesezeiger*. Nach jeder Operation wird er eine angemessene Anzahl von Bytes weiter bewegt.

Während bei Textdateien diese Position mühsam gesucht werden muß, weiß Pascal bei typisierten Dateien genau, wie groß so ein Schritt ist. Sie haben es bei der Vereinbarung der Dateivariablen ja selbst angegeben. Damit sind die Zugriffe viel schneller und effektiver – für Daten fester Länge!

Beispiel für typisierte Dateien

Im folgenden Beispiel sollen immer Blöcke von 512 Byte gelesen werden – eine magische Größe für den effektiven Dateizugriff. Der Inhalt der Datei soll nicht interessieren, für uns sind das nur Bytes.

Der eigene Datentyp wird als Feld von 512 Zeichen definiert. Genaueres über die Definition von Feldern (Arrays) erfahren Sie im nächsten Kapitel.

```
program BlockLeser;

type
    TBlock512 = Array [1..512] of Byte;

var
    datei: File of TBlock512;
    daten: TBlock512;

begin
    Assign(datei, 'C:\COMMAND.COM');
    Reset(datei);

    while not Eof(datei) do
        begin
            Read(datei, daten);
```

```
            { Hier folgen die Verarbeitungsschritte ...}
            { ... }
        end;

    Close(datei);
end.
```

Listing 6.2: Beispiel für die Verarbeitung einer Datei fester Satzlänge (6_2.PAS)

Wenn Sie sich das Beispiel ansehen, wird Ihnen auffallen, daß es eigentlich kaum einen Unterschied zum vorherigen Beispiel mit der Textdatei gibt.

Typisierte Textdateien

Kann man mit typisierten Dateien auch Textdateien lesen? Man kann, allerdings auf eine umständliche Art. Haben Ihre Sätze alle die gleiche Länge, dann ist es nicht weiter kompliziert:

```
program FesteZeilenlänge

type
    TZeile = string[20];

var
    datei : File of TZeile;
    zeile: TZeile;
    ...
```

Weiter möchten wir auf diese Typvereinbarung für Zeichenketten noch nicht eingehen, im Kapitel über benutzerdefinierte Datentypen erfahren Sie mehr darüber.

Die andere Alternative wäre, eine Datei mit der kürzesten Satzlänge zu vereinbaren. Für reine Bytes ist das der vordefinierte Typ `byte`, ansonsten nehmen Sie für die Zeichen den Typ `char`. Dann können Sie die Datei zeichenweise lesen!

Das ist allerdings erheblich langsamer, als wenn Sie mit `Text` arbeiten. Aber es gibt auch Fälle, wo man solch eine Art der Verarbeitung effektiv nutzen kann. Die Entscheidung darüber liegt einzig bei Ihnen und den Bedürfnissen des Programms:

```
program ZeichenLeser;

 var
   datei: File of char;
   daten: char;

begin
   Assign(datei, 'C:\AUTOEXEC.BAT');
   Reset(datei);

   while not Eof(datei) do
        begin
                Read(datei, daten);
                WriteLn(daten);
        end;

   Close(datei);
end.
```

Listing 6.3: Beispiel für die zeichenweise Verarbeitung einer Datei (6_3.PAS)

6.1.4 Untypisierte Dateien

Wenn sich das Programm keinerlei Gedanken um den Inhalt und die Datenstrukturen in einer Datei machen muß, handelt es sich aus seiner Sicht um eine untypisierte Datei. Die Deklaration der dazu passenden Variable ist denkbar einfach:

```
var
    datei: File;
...
```

6.1.5 Lesen und Schreiben einer Datei

Sie kennen die Befehle, welche wir Ihnen hier präsentieren, schon aus den vorangegangenen Abschnitten des Kapitels. Trotzdem möchten wir ihre Verwendung noch einmal zusammenfassen, und den einen oder anderen nützlich Tip geben.

Dateifunktionen

Dabei geht es um folgende Funktionen und Prozeduren:

Tabelle 6.1: Elementare Dateifunktionen

Funktion/Prozedur	Bedeutung
Assign (fvar, name);	Verbindung zwischen Dateivariable und Name herstellen.
Reset (fvar)	Die Datei öffnen bzw. an deren Anfang gehen, wenn die Datei nicht existiert: Fehler.
ReWrite (fvar)	Die Datei auf die Länge Null bringen, was vor allem beim Überschreiben wichtig ist. Sollte die Datei noch nicht existieren, wird sie angelegt.
Close (fvar)	Eine geöffnete Datei schließen. Freigabe der begrenzten Anzahl an Verwaltungsinformationen.
Eof (fvar)	Test, ob Dateiende erreicht ist.
Read (fvar, ...)	Lesen aus der Datei.
Write (fvar, ...)	Schreiben in die Datei.

Öffnen, Anlegen oder Leeren?

Um eine Datei zu öffnen, müssen Sie in zwei Schritten vorgehen:

- Verbinden Sie die Dateivariable mit einem Dateinamen. Dazu dient `Assign()`.

- Öffnen Sie die Datei. Hier kann `Reset()` genutzt werden, was bei einer existierenden Datei dazu führt, daß der Dateizeiger auf den ersten Satz positioniert wird. Existiert die Datei nicht, kommt es zu einer Fehlermeldung.

```
program AltDat;

var
     datei: Text;

begin
     Assign(datei, 'DATEI.TXT');
     Reset(datei);
     ...
```

- Öffnen Sie dagegen die Datei mit `ReWrite()`, wird sie neu angelegt, wenn sie nicht existiert. Eine existierende Datei wird dagegen auf die Länge Null gebracht. Das entspricht dann im Prinzip einem Überschreiben, wenn später in die Datei hineingeschrieben wird.

```
program NeuDat;

var
     datei: Text;

begin
     Assign(datei, 'DATEI.TXT');
     ReWrite(datei);
...
```

FileMode: Eine Datei nur zum Lesen öffnen

Die Variable `FileMode` bestimmt den Zugriffscode, der an DOS weitergegeben wird, wenn typisierte und untypisierte Dateien mit `Reset()` geöffnet werden.

Der Vorgabewert von `FileMode` ist 2. Zuweisen eines anderen Wertes an `FileMode` bedeutet, daß alle folgenden Aufrufe der Prozedur `Reset()` diesen Modus verwenden. Der Gültigkeitsbereich der `FileMode`-Werte hängt von der verwendeten DOS-Version ab. Bei allen Versionen gelten diese Modi:

Tabelle 6.2: Modi beim Öffnen der Dateien

Modus	Bedeutung
0	Read only (Nur Lesen)
1	Write only (Nur Schreiben)
2	Read/Write (Lesen und Schreiben)

Um eine Datei nur zum Lesen zu öffnen, müssen Sie vorher den Wert in `FileMode` neu setzen:

```
program NurLesen;

var
    datei: Text;
    merker: integer;

begin
    merker := FileMode;

    FileMode := 0;
    Assign(datei, 'DATEI.TXT');
    ReWrite(datei);
    FileMode := merker;
    ...
```

Die Datei kann jetzt nur noch gelesen werden, ein versehentliches Überschreiben ist damit unmöglich geworden!

Lesen aus der Datei

Wenn Sie aus der Datei lesen, machen Sie das mit `Read()` oder `ReadLn()` für Textdateien. Wie auch bei der Eingabe von der Tastatur können Sie bei `Read()` durchaus mehrere Zielvariablen angeben.

```
Read(datei, variable);
Read(datei, var1, var2, var3 ...)
```

Auf alle Fälle müssen Sie testen, ob das Dateiende vielleicht schon erreicht wurde, denn wenn Sie über das Ende einer Datei hinaus zu lesen versuchen, wird das Programm mit einer Fehlermeldung abbrechen.

Der einfache Test erfolgt mit der Funktion `Eof(dateivariable)`, die einen Wert `true` oder `false` zurückgibt, je nachdem, ob das Dateiende erreicht wurde oder nicht.

Meistens benutzt man sie im Rahmen einer Schleifensteuerung, mit der eine Datei satzweise ausgelesen wird.

```
while not Eof(datei) do
    begin
        ...
```

Schreiben in eine Datei

Normalerweise schreibt man in eine Datei an deren bisheriges Ende. Die neuen Informationen werden also angehängt. Um die damit zusammenhängende Verwaltung des Plattenplatzes kümmert sich das Betriebssystem. Aber dafür müssen Sie sich erst einmal ans Dateiende positioniert haben!

Wenn Sie noch irgendwo in der Mitte der Datei stehen und dann `Write()` oder `WriteLn()` aufrufen, werden alle Inhalte der Datei ab der aktuellen Position des Dateizeigers mit den neuen Werten überschrieben! Wenn Sie etwas Neues schreiben, wird die Datei also nicht ausgeweitet, sondern überschrieben.

In diesem Zusammenhang sind `Seek()` und `FilePos()` von Interesse, mit denen eine Positionierung und Orientierung innerhalb der

Datei möglich ist, sowie `Truncate()`, mit dem eine Datei auf die Länge Null gebracht werden kann.

Die Verwendung von `Write()` und `WriteLn()` funktioniert übrigens genauso, wie Sie es bisher für die Ausgabe auf dem Bildschirm gewöhnt waren.

Datei schließen

Es gibt unter MS-DOS nur eine recht begrenzte Zahl von Verwaltungseinheiten für geöffnete Dateien. Im Zusammenspiel mit Windows kann man da schnell an kritische Grenzen stoßen.

Dann kann solange keine weitere Datei mehr geöffnet werden, bis wieder einige dieser Verwaltunginformationen freigegeben wurden, quasi recycelt werden können.

Daher sollten Sie sich angewöhnen, eine Datei immer zu schließen, wenn Sie sie für längere Zeit oder gar nicht mehr im Programm benötigen.

Der Aufruf dazu ergeht an das Betriebssystem mit `Close()`, dem Sie den Namen der Dateivariablen mitgeben.

6.1.6 Eine eigene Fehlerbehandlung

Da beim Versuch, ein `Reset()` auf eine nicht existierende Datei anzuwenden, automatisch ein Fehler generiert wird, und das Programm abbricht, müssen Sie manchmal die Fehlerbehandlung selbst in die Hand nehmen.

Abschalten der automatischen Fehlerbehandlung

Muß die automatische Fehlerbehandlung – zumindest kurzfristig – ausgeschaltet werden, setzen Sie die Compilerdirektive `{$I}` ein.

Nun können Sie die Fehlerbehandlung selber steuern. Laut Online-Hilfe müssen, bevor es mit dem Programmablauf weitergehen kann, erst einmal alle nötigen Tests in Ihrem Programm vorgenommen worden sein:

»Im Modus {$I+} codiert der Compiler nach jeder Ein- und Ausgabe den Aufruf einer Prüfroutine, die das Programm im Fehlerfall mit einer entsprechenden Meldung abbricht. Im Modus {$I-} unterbleibt diese automatische Prüfung: hier muß das Programm mögliche Fehler über den Aufruf von IOResult selbst prüfen. Nach einem Fehler werden folgende Ein- und Ausgabebefehle solange ignoriert, bis das Programm IOResult aufgerufen hat. Jeder Aufruf von IOResult setzt die intern gespeicherte Fehlernummer auf 0 zurück.«

Beispiel

Durch diesen einfachen Mechanismus ist es beispielsweise möglich, ein kleines Programm zu schreiben, das testet, ob eine Datei existiert:

```
program DateiTest;

var
   datei: File;
   name: string;
   fehler: integer;

begin
   Write('Den Dateinamen: '); ReadLn(name);
   Assign(datei, name);

   {$I-}
   Reset(datei);
   fehler := IOResult;
   Close(datei);
   {$I+}

   Write('Die Datei ', name, ' existiert');
   if (fehler <> 0) and (name <> '') then
        WriteLn(' nicht');
   WriteLn('!');
end;
```

Listing 6.4: Test auf die Existenz einer Datei (🖫 6_4.PAS)

6.1.7 Ein abschließendes Beispiel

Zum Abschluß soll Ihnen ein weiteres Beispiel die Verwendung der Funktionen noch einmal mit Hilfe der Textdateien zeigen. Es handelt sich schon um ein etwas größeres Programm, da wir hier auch ausnahmsweise einmal etwas mehr Fehlerbehandlung hineingesteckt haben. Beachten Sie dabei vor allem die Verwendung der Compileroption `{$I}`, mit der wir in der Lage sind, unsere eigene Fehlerbehandlung »scharf« zu machen.

Ein Textkopierer

Bei dem Beispiel handelt es sich um eine Art Kopierprogramm, welches aber wirklich nur für Textdateien zu gebrauchen ist. Es erfragt vom Benutzer die Namen zweier Dateien, öffnet sie, und überträgt den Inhalt der einen Datei in die andere Datei.

Beachten Sie bitte, daß sich ein echtes Kopierprogramm nur darin unterscheidet, daß es nicht mit Texten, sondern mit Bytes – natürlich in größeren Blöcken fester Größe – arbeitet:

```
 1 program TextKopierer;
 2
 3 uses
 4     Dos;
 5
 6 var
 7    quelle, ziel : Text;
 8    zeile : string;
 9    qname, zname : string;
10
11 begin
12
13   { Erst einmal die Dateinamen erfragen }
14   Write('Den Namen der Quelldatei: '); ReadLn(qname);
15   Write('Den Namen der Zieldatei: ');  ReadLn(zname);
16
17   { Zugriff vorbereiten }
18   Assign(quelle, qname);
```

```
19
20  { Zugriff ausführen, vorher die offizielle
21    Fehlerbehandlung abschalten, danach anschalten }
22  {$I-}
23  Reset(quelle);
24  {$I+}
25
26  { Fehlerbehandlung ausführen, evtl. Programmabbruch }
27  if IOResult <> 0 then
28     begin
29          WriteLn('Quelle existiert nicht, Abbruch!');
30          Halt(1);
31     end;
32
33  { Zugriff vorbereiten }
34  Assign(ziel, zname);
35
36  { Zugriff ausführen, vorher die offizielle
37    Fehlerbehandlung abschalten, danach anschalten }
38  {$I-}
39  Reset(ziel);
40  {$I+}
41
42  { Fehlerbehandlung ausführen, evtl. Abbruch, ansonsten
43    Neuanlage der Zieldatei }
44  if IOResult = 0 then
45     begin
46          WriteLn('Ziel existiert schon, Abbruch!');
47          Halt(2);
48     end
49  else
50     ReWrite(ziel);
51
52  { Die Kopieroperation: Lese aus Quelle ins RAM
53                         Schreibe aus RAM ins Ziel }
54  while not Eof(quelle) do
55        begin
```

```
56              ReadLn(quelle, zeile);
57              WriteLn(ziel, zeile);
58          end;
59
60    { Fertig, alles wieder schließen }
61    Close(quelle);
62    Close(ziel);
63 end.
```

Listing 6.5: Ein Textkopierer (🖫 6_5.PAS)

Das Programm ist, auch wenn es auf den ersten Blick nicht so aussieht, recht primitiv:

- Die Quelldatei wird geöffnet, ohne daß wir es extra sagen müßten, wie immer mit `Read/Write`. Dann wird die Zieldatei geöffnet. Wenn sie schon existiert, soll das Programm sicherheitshalber abgebrochen werden, ansonsten wird die Datei angelegt. Auch sie ist standardmäßig mit `Read/Write` geöffnet worden.
- Wenn die Quelldatei nicht existiert, bricht das Programm ab, was soll man sonst auch schon machen? Bei der Zieldatei hätten wir die Möglichkeit gehabt, direkt mit `ReWrite()` zuzuschlagen. Das hätte fast nie einen Fehler gegeben. Allerdings wäre dann eine existierende Datei gnadenlos überschrieben worden, da `ReWrite()` die Angewohnheit hat, existierende Dateien auf die Länge Null zu bringen und ansonsten die Datei neu anzulegen. Und das wollten wir nicht, denn existierende Dateien sollten verschont werden. Daher »testen« wir mit `Reset()`, ob es zu einem Fehler kommt. Wenn ja, gibt es noch keine Datei gleichen Namens und ein `ReWrite()` kann bedenkenlos angewandt werden. Ansonsten lassen wir die Datei in Ruhe.
- Der Kopiervorgang geschieht so, daß erst einmal der Satz aus der Quelle gelesen wird und im Speicher des Programms bleibt. Von dort wird er ins Ziel übertragen. Leider geht eine direkte Zuweisung der Art `quelle := ziel` nicht. Sie müssen statt dessen immer den Weg über eine Zwischenspeicherung im RAM gehen.

6.1.8 Interessantes aus der Unit System

Nun wissen Sie also über die wichtigsten Dateitypen und etliche grundlegende Dateifunktionen in Pascal Bescheid. Die nachfolgende Tabelle soll Ihnen noch Hinweise auf einige interessante Funktionen und Prozeduren geben. Allerdings ist sie nicht umfassend, und so sollten Sie auf alle Fälle die Handbucheinträge über die `Unit System` konsultieren!

Tabelle 6.3: Interessante Dateifunktionen

Funktion/Prozedur	Bedeutung
Erase (fvar)	Eine Datei löschen.
Rename (fvar, neuname)	Eine Datei umbenennen.
Truncate (fvar)	Eine Datei ab der aktuellen Position des Dateizeigers abschneiden.
Seek (fvar, nr)	Eine Position innerhalb der Datei suchen.
FilePos (fvar)	Liefert die aktuelle Position (Satznummer), auf die der Dateizeiger verweist.
FileSize (fvar)	Liefert die aktuelle Dateigröße.
ChDir (verz)	Wechselt das aktuelle Verzeichnis des aktuellen Laufwerks.
MkDir (name)	Legt ein neues Verzeichnis an.
GetDir (k, name)	Ermittelt den Namen eines aktuellen Verzeichnisses; die Kennung k identifiziert das Laufwerk.
RmDir (name)	Löscht ein Verzeichnis, das leer sein muß.

6.2 MS-DOS läßt grüßen

Bevor wir dieses Kapitel abschließen, lassen Sie uns noch einen kurzen Blick auf die Möglichkeiten werfen, die Ihnen Pascal bietet, um mit den Eigenheiten von MS-DOS zurechtzukommen.

6.2.1 Zusätzliche Funktionen in Borland-Pascal

Dazu muß man vorab sagen, daß die ursprüngliche Entwicklung von Pascal kein MS-DOS kannte, Pascal ist mindestens 5 Jahre älter. Daher sind alle Funktionalitäten, die auf DOS zugreifen, durch Funktionen und Prozeduren realisiert, die Borland seinen Pascal-Dialekten mitgegeben hat.

Funktion -> Interrupt -> Betriebssystem

Ihrerseits rufen diese Funktionen und Prozeduren wieder Assemblercode oder über die verschiedenen *Interrupts* direkt das Betriebssystem auf.

Je nach der Entwicklung, die DOS/Windows und Pascal nehmen, werden diese Funktionen uns vermutlich noch länger erhalten bleiben, garantiert ist das jedoch auf Dauer nicht. Daher sollten Sie solche betriebssystembezogenen Teile nach Möglichkeit vom restlichen Programmcode abtrennen, um sie später besser gegen andere Verfahren austauschen zu können, ohne gleich das ganze Programm neu schreiben zu müssen.

Hier wäre also ein sinnvoller Ansatzpunkt für die im vorangehenden Kapitel beschriebenen Funktionen, Prozeduren und Units!

Tabelle 6.4: Funktionen für das MS-DOS-Dateisystem

Funktion/Prozedur	Aufgabe
SetFTime (Prozedur)	Setzt Datum und Uhrzeit der letzten Veränderung einer Datei.
GetFTime (Prozedur)	Ermittelt Datum und Uhrzeit der letzten Veränderung einer Datei.
DiskFree (Funktion)	Liefert die Größe des freien Speicherplatzes auf der angegebenen Diskette/Platte zurück.
DiskSize (Funktion)	Liefert die Gesamtkapazität in Byte der angegebenen Diskette/Platte zurück.
GetVerify (Prozedur)	Liefert den Status des Verify-Flags von DOS zurück.
SetVerify (Prozedur)	Setzt das Verify-Flag von DOS.
FExpand (Funktion)	Erweitert einen Dateinamen in einen qualifizierten Dateinamen (Laufwerk, Verzeichnis und Name).
FSearch (Funktion)	Sucht nach einer Datei.
FSplit (Funktion)	Zerlegt einen vollständigen Dateinamen in seine drei Bestandteile.
FindFirst (Prozedur)	Sucht ein angegebenes Verzeichnis nach dem ersten Vorkommen eines Dateinamens ab.
FindNext (Prozedur)	Sucht das nächste Vorkommen eines Verzeichniseintrags, das dem Namen und den Attributen des letzten Aufrufs von FindFirst entspricht.
GetFAttr (Prozedur)	Liefert die Attribute einer Datei zurück.
SetFAttr (Prozedur)	Setzt die Attribute einer Datei.

Wie werden die Funktionen eingesetzt?

Sie setzen die obigen Funktionen ein, wie alle anderen auch. Nichts deutet darauf hin, daß sie etwas Besonderes wären. Sie müssen lediglich daran denken, daß auch die passende Unit in Ihr Programm mit eingebunden wird. Bei einem MS-DOS-Programm heißt die Unit `Dos`, bei einem Windows-Programm `WinDos`.

Alle weiteren Angaben zu den Funktionen entnehmen Sie bitte den Handbüchern zum Compiler bzw. der Online-Hilfe.

6.2.2 Beispiel: Ermittlung der Plattenauslastung

Am besten wird es sein, wenn wir Ihnen den Einsatz der MS-DOS-spezifischen Funktionen an einem kleinen Beispiel zeigen.

DiskSize und DiskFree

Es geht dabei darum, einige Informationen über die Festplatte zu erlangen. Beispielsweise stellt sich ja manchmal die Frage, wieviel Platz noch frei ist und wie der Auslastungsgrad des Datenträgers denn ist (Verbrauchter Platz : Gesamter Platz).

Dazu setzen wir die Funktionen `DiskSize()` für die Ermittlung der Gesamtkapazität der Platte und `DiskFree()` für die Bestimmung des noch freien Platzes ein. Wenn Sie in der Hilfe nachlesen, werden Sie feststellen, daß beide Funktionen einen Rückgabetyp `longint` haben. Das ist nötig, da moderne Festplatten recht groß werden können und der Ergebniswert in Byte angegeben wird.

Sollte ein Fehler auftreten, erhalten Sie den Wert `-1` zurück, ansonsten einen positiven Zahlenwert. Das macht Sinn, da bisher noch keine Laufwerke mit negativer Kapazität bekannt sind.

Als Argumente beim Aufruf erwarten beide Funktionen eine Laufwerksnummer. Der Wert 0 steht für das aktuelle Laufwerk, alle Folgewerte stehen für die anderen Laufwerke des Rechners (A = 1, B = 2, C = 3 ...).

Das Programm

Die gesamte Auslastung der Platte läßt sich dann mit dem folgenden kleinen Programm recht einfach ermitteln:

```
 1 program DiskUsage;
 2
 3 uses
 4     Dos;
 5
 6 var
 7     kapazitaet : longint;
 8     benutzt : longint;
 9     nummer : byte;
10     laufwerk: char;
11
12 begin
13     { Laufwerksbuchstaben entgegennehmen }
14     Write('Bitte den Laufwerksbuchstaben angeben : ');
15     ReadLn(laufwerk);
16
17     { Umwandeln in die zugeordnete Laufwerksnummer 1, 2, }
18
19     nummer := Ord(UpCase(laufwerk)) - Ord('A');
20
21     { Test auf Eingabefehler, Abbruch mit Errorlevel 1 }
22     if (nummer < 1) or (nummer > 26) then
23       begin
24           WriteLn('Laufwerk was ??? -> ', laufwerk);
25           Halt(1);
26       end;
27
28     { Ermittlung der Plattenkapazität }
29     kapazitaet := DiskSize(nummer);
30
31     { Test auf Fehler (-1) wegen ungültigem Laufwerk,
32       Abbruch mit Errorlevel 2 }
```

```
33      if k < 0 then
34        begin
35             WriteLn('Für Laufwerk ', laufwerk,
36                     ' keine Angaben möglich!');
37             Halt(2);
38        end;
39
40      { Ermittlung benutzer Bytes = Kapazität-Freie Bytes}
41      benutzt := kapazität - DiskFree(nummer);
42
43      { Berechnung und Ausgabe des Auslastungsgrads }
44      WriteLn(((benutzt / kapazitaet) * 100):4:4,
45              ' % Auslastung');
46
47      { Programmende mit Errorlevel 0 = alles OK }
48      Halt(0);
49 end.
```

Listing 6.6: Bestimmung der Plattenauslastung

Das Programm fragt vom Benutzer ab, welches Laufwerk bearbeitet werden soll. Da es nicht zumutbar ist, daß der Benutzer eine Laufwerksnummer eingibt, darf er einen Buchstaben angeben. Der wird für alle Fälle auf Großbuchstabe getrimmt und dann in einen Wert umgewandelt. Da im ASCII-Alphabet der Buchstabe `'A'` nicht an erster Stelle steht, muß noch eine kleine Wertkorrektur vorgenommen werden.

Sollte danach nicht ein Wert von `'A'` bis `'Z'` vorliegen, hat der Benutzer wohl einen Fehler gemacht, und das Programm bricht ab. Ansonsten wird die Kapazität des Laufwerks ermittelt. Sollte da ein `-1` zurückkommen, wurde wieder ein Fehler gemacht, und die Konsequenz ist wiederum ein Programmabbruch.

Nun wird die Anzahl der benutzten Bytes bestimmt, und daraus der Grad der Belegung errechnet. Der entsprechende Prozentwert wird ausgegeben und das Programm friedlich beendet.

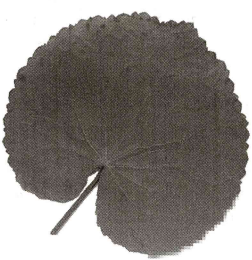
Teile und Herrsche 7

7.1 Was sind Subroutinen?..... 186

7.2 Definition einer Subroutine 192

7.3 Variablen und Subroutinen..... 203

7.4 Die Definition von Units..... 215



Unsere bisherigen Programme waren alle recht klein und unscheinbar. Was vor allem daran liegt, daß sie keine Aufgabenstellungen aus der wirklichen Welt lösen, sondern immer nur Teilausschnitte der Programmiersprache verdeutlichen sollten.

Programmieraufgaben aus der wirklichen Welt zeichnen sich durch eine enorme Größe und Komplexität aus. Zigtausende von Programmzeilen müssen entworfen und geschrieben werden. Wenn man das alles nur in den kleinen Bereich zwischen `begin` und `end` eines Hauptprogramms schreibt, wird die Unübersichtlichkeit schnell so groß, daß das Programm eher mit Glück als mit Plan an ein Ende kommt.

Außerdem werden bestimmte Programmteile immer wieder vorkommen. Natürlich könnte man sie mit KOPIEREN und WIEDEREINFÜGEN nutzen, aber was ist, wenn Änderungen in allen Kopien dieses Codestücks nötig werden?

Während man in vielen älteren Programmiersprachen diesem Phänomen hilflos ausgeliefert ist, gibt es bei Pascal Mittel und Wege, um diese ausufernden Programmklötze wieder in den Griff zu bekommen und besser zu gliedern. Es handelt sich um das Konzept der Subroutinen, die Teilaufgaben erledigen, und später wie Softwarebausteine eingesetzt werden können. Außerdem lassen sie sich in speziellen Dateien sammeln und so gewissermaßen als *Softwarebibliotheken (Unit)* einsetzen.

Dieses Kapitel zeigt Ihnen die Definition und Verwendung der Subroutinen, ihre kleinen Haken und Ösen und vor allem auch, wie Sie sich eigene Units anlegen können.

7.1 Was sind Subroutinen?

Aus den vorangegangenen Erläuterungen läßt sich ganz gut schließen, wozu *Subroutinen* dienen. Es sind Programmbausteine, die eine einzige kleine Teilaufgabe möglichst vollständig und korrekt erledigen. Aus diesen Subroutinen und weiterem verbindenden Code werden dann die Programme zusammengesetzt.

7.1.1 Funktionen und Prozeduren

Pascal unterscheidet zwei Arten von Subroutinen. Der große Unterschied liegt darin, ob sie Ergebniswerte zurückgeben oder nicht. Die beiden Arten von Subroutinen sind:

- *Funktionen*
- *Prozeduren*

Allen beiden ist gemeinsam, daß sie abgeschlossene Einheiten sind und eventuell bei ihrem Aufruf noch weitere Informationen benötigen, die in Form von Parametern mitgegeben werden.

Funktionen

Funktionen sind Subroutinen, die in Ausdrücken eingesetzt werden und einen Ergebniswert zurückliefern. Vergleichbar sind sie den mathematischen Funktionen. Ein Beispiel dafür ist die vordefinierte Funktion zum Ziehen von Wurzeln:

```
function Sqrt(X: Real): Real;
```

Diese Funktion erwartet als Parameter (Argument) einen `real`-Wert. Dieser kann eine Konstante, der Inhalt einer Variablen oder ein Funktionsergebnis sein. Die Nutzung zeigt das folgende Beispiel:

```
 1 program Wurzel;
 2
 3 var
 4     eingabe, ergebnis : real;
 5
 6 begin
 7     Write('Eine Zahl bitte: ');
 8     ReadLn(eingabe);
 9
10     ergebnis := Sqrt(eingabe);
11
12     WriteLn('Wurzel aus ', eingabe, 'ist ', ergebnis);
13 end.
```

Listing 7.1: Beispiel für einen Funktionsaufruf (7_1.PAS)

Prozeduren

Prozeduren zeichnen sich dadurch aus, daß sie keinen Rückgabewert haben. Es besteht zwar die Möglichkeit, daß Variablen, welche als Parameter übergeben werden, eine Änderung erfahren, aber einen Prozedurwert, analog zum Funktionswert, haben Prozeduren eben nicht. Daher werden sie auch nicht in Ausdrücken eingesetzt.

Einige Vertreter der Prozeduren sind Ihnen sicherlich wohlbekannt: `WriteLn()`, `Write()`, `ReadLn()` und `Read()`.

Der Entwurf

Wie kommt man nun zu seinen Subroutinen. Die Antwort ist ganz einfach: durch sorgfältigen Programmentwurf. Und natürlich durch Beobachtung. Der Programmentwurf führt dazu, daß sich aus der zuerst unfaßlichen Gesamtaufgabe kleine Einheiten herauskristallisieren.

Diese werden noch weiter zerlegt und verfeinert, bis schließlich kleine, funktional abgeschlossene Einheiten geringer Komplexität übrigbleiben. Das sind dann die Subroutinen. Eine Faustregel besagt, daß eine gute Subroutine tunlichst nicht länger wird, als Programmtext auf eine DIN A4-Seite paßt.

Daneben fällt einem beim Programmieren sicherlich auf, wenn bestimmte Codeteile in der einen oder anderen Form immer wiederholt werden. Dann hat man einen Kandidaten für eine Subroutine vorliegen, der beim vorangehenden Entwurf wohl übersehen wurde. Nun ist eine gute Gelegenheit, dies nachzuholen.

Diese Aufteilung des Programmprojekts in kleinere Einheiten bietet so viele Vorteile, daß Sie sich immer angewöhnen sollten, vor dem »Loshacken« erst einmal über die Aufgabe und ihre Ziele, das benötigte Wissen und die mögliche Aufteilung in Subroutinen nachzudenken. Und das darf ruhig etwas länger dauern. Je mehr Zeit Sie für diesen Entwurf verwenden, desto mehr Spaß macht später die Ausführung der Programmierung. Dann kann man sich nämlich auf das konzentrieren, was Spaß macht: das »Hacken«. So bekommen Sie auch große Aufgabenstellungen wieder in den Griff. Und daher auch die Kapitelüberschrift: »Teile und Herrsche«.

7.1.2 Bibliotheken

Da viele Aufgabenstellungen in verschiedensten Programmen immer wieder vorkommen, lohnt es sich, die dazu passenden Subroutinen in eigenen Dateien, den Units, zu sammeln, und sie so immer wieder zu benutzen.

Einbindung der Bibliotheken

Bibliotheken müssen dem Compiler gewissermaßen angekündigt werden. Dies muß beispielsweise schon deshalb sein, damit er die aus den Units stammenden Namen von Subroutinen oder gemeinsamen Variablen dort nachsehen kann, wenn sie im Hauptprogramm benutzt werden.

Um eine Unit einzubinden, können Sie, wie schon in Kapitel 1 gesagt wurde, den Befehl `uses` benutzen:

```
program TestProgramm;

uses
    WinCrt, WinDos;
```

Manchmal kann es vorkommen, daß die Unit in einem anderen Verzeichnis liegt als in dem, wo Sie gerade arbeiten. Nun besteht die Möglichkeit, den Compiler so zu konfigurieren, daß er in bestimmten Verzeichnissen immer nachschaut, wenn eine Unit angegeben wird. In diesem Fall brauchen Sie bei der Einbindung der Unit nichts weiter anzugeben.

Nur für den Fall, daß die Unit sonst wirklich nicht zu finden wäre, besteht die Möglichkeit, das Verzeichnis in dem sie liegt, und wie die dazugehörige Datei heißt, anzugeben:

```
program TestApp;

uses Unit3 in 'C:\TMP\UNIT3.PAS';

begin
```

Vordefinierte und selbstdefinierte Bibliotheken

Pascal bietet schon eine ziemliche Anzahl an vordefinierten Subroutinen an. Sie sind in verschiedene Units (*Standardunits*) aufgeteilt. Diese Aufteilung erfolgte nicht willkürlich, sondern so, daß logisch zusammengehörige Aufgabengebiete und ihre Subroutinen eine Unit ergeben.

Tabelle 7.1: Standardbibliotheken

Unitname	**Einsatzbereich**
Crt	Direkte Kontrolle des Bildschirms, der Tastatur, der Farben und des Tongenerators
Dos	Routinen für das Betriebssystem und zur Dateiverarbeitung
Graph	Grafikpaket mit mehr als 50 Routinen von bitorientierten Prozeduren
Graph3	Implementation der Standard- und Turtlegrafik der Version 3.0
Overlay	Prozeduren, Funktionen und Symbole, die von Borland Pascals Overlay-Verwaltung benutzt werden
Printer	Definiert eine Textdatei namens *LST* und bindet sie an den Druckerport LPT1
System	Laufzeitbibliothek von Borland Pascal - SYSTEM.TPW Windows - SYSTEM.TPP DOS Protected-Mode - SYSTEM.TPU DOS Real-Mode
Turbo3	Für die Übernahme von Programmen der Version 3.0 gedacht
WinDOS	Routinen für das Betriebssystem und zur Dateiverarbeitung
Strings	Unterstützt eine neue Klasse von Zeichenstrings, die null-terminierten Strings

Tabelle 7.1: Standardbibliotheken (Fortsetzung)

WinAPI	Diese API-Funktionen von Windows 3.1 werden von dem Protected-Mode-Laufzeitsystem von Borland unterstützt
WinCrt	Implementiert in einem Fenster ein Textfenster, das einem herkömmlichen Textterminal entspricht
WinPrn	Die Unit WinPrn erlaubt es, die Ausgaben eines unter Windows laufenden Programms an einen beliebigen Drucker weiterzuleiten
WinProcs	Definiert Funktions- und Prozedurkopfzeilen für Windows API
WinTypes	Definiert Borland-Pascal-Versionen aller Typen, welche von Windows-API-Routinen verwendet werden

Diese Standard-Units bilden den Kern eines Pascal-Programms, und sind – bis auf die Unit System – in der Datei *TPW.TPL* (für Windows), *TPP.TPL* (für *Protected-Mode*) oder *TURBO.TPL* (für *Real-Mode*) gespeichert. Durch Angabe der Unit weiß der Pascal-Compiler, daß er diese *TPL*-Datei zu laden hat.

Neben den Standardbibliotheken des Compilers gibt es eine riesige Zahl von vorgefertigten Units, die sowohl kommerziell als auch auf freier Basis vertrieben werden. Der Datenträger zum Buch enthält eine Auswahl davon. Alle Units dieser Welt mitzugeben, wäre unmöglich, denn keiner weiß, wie viele das sein mögen!

Und dann kommen noch Ihre selbstdefinierten Bibliotheken hinzu. Sie können so eigene professionelle Bibliotheken aufbauen, die vielleicht sogar kommerziell einsetzbar sind. Wie das genau geht, erfahren Sie zum Ende dieses Kapitels.

Projekte

Ein Programm setzt sich damit aus den Bibliotheken, Units und dem Hauptprogramm zusammen. Um den Überblick zu bewahren, bietet der Borland Compiler eine Projektverwaltung an, in der Sie die einzelnen Komponenten des Programms eintragen.

Sollten sich Änderungen an einer oder mehreren dieser Komponenten ergeben, merkt sich die Projektverwaltung dieses und berücksichtigt beim nächsten Übersetzungslauf alle direkt und indirekt von der Änderung betroffenen Dateien.

So wird sichergestellt, daß alle Dateien auf ihrem aktuellen Stand sind und das Programm immer vollständig übersetzt wird. Daraus ergibt sich übrigens noch ein weiterer Vorteil der Zerlegung eines Programms in eine Hauptdatei und mehrere Units: Alle Units, die nicht verändert wurden, müssen auch nicht neu übersetzt werden! Und damit können gerade Änderungen an großen Programmprojekten sehr viel schneller neu übersetzt werden.

7.2 Definition einer Subroutine

Kommen wir nun zur Definition der Subroutinen. Neben der Angabe der Namen, eventuell vorhandener Parameter und Rückgabetypen ist natürlich der Prozedurrumpf anzugeben. Alles in allem kann man aber sagen, daß es nicht sehr kompliziert ist, eine Prozedur oder eine Funktion zu definieren.

7.2.1 Allgemeines

Bei der Definition einer Subroutine gibt es bezüglich der Reihenfolge zwei Dinge, die Sie berücksichtigen müssen.

Bild 7.1: Syntaxdiagramm für Subroutinen (Beispiel: procedure)

Der Ort der Definition

Es müssen erst alle Funktionen und Prozeduren definiert sein, bevor das Hauptprogramm – wie immer zwischen `begin` und `end.` eingepackt – ablaufen kann. Das hängt damit zusammen, daß eine Subroutine grundsätzlich vor ihrer ersten Benutzung bekannt sein muß:

```
 1 program Beispiel;
 2
 3 (*-- Beginn der Subroutinen --*)
 4 procedure proz1(i : integer);
 5     begin
 6           ...
 7     end;
 8
 9 procedure proz2(i : real);
10     begin
11           ...
12     end;
13
14 (*-- Beginn des Hauptprogramms --*)
15 begin
16     ...
17 end.
```

Da solche Sammlungen schnell unübersichtlich werden, packt man die Subroutinen oftmals in eine eigene Unit, während das Hauptprogramm aus dem Einbinden der Unit und dem Aufruf der Funktionen besteht. Dem Hauptprogramm kommt somit wirklich nur noch eine steuernde Funktion zu:

```
1 program Beispiel;
2
3 (*-- Einbindung der eigenen Unit --*)
4 uses
5     SubUnit;
6
```

```
 7 begin
 8     ...
 9 end.
```

Vorabdeklarationen

Es gibt nur einen einzigen Fall, wo die Sache etwas komplexer gerät. Und zwar, wenn eine Subroutine Bezug auf eine andere Subroutine nimmt. Ein Phänomen, das beim Einsatz von Units oft vorkommt.

```
 1 { Aufrufende Prozedur }
 2 procedure proz1(i : integer);
 3     begin
 4             ...
 5             proz2(wert);
 6             ...
 7     end;
 8
 9 { Aufgerufene Prozedur }
10 procedure proz2(i : real);
11     begin
12             ...
13     end;
```

Es gilt nämlich die Regel, daß zumindest der Kopf jeder Subroutine vor ihrer ersten Benutzung bekannt sein muß. Die Lösung dafür ist, daß eine *Forward-Deklaration* der Subroutine gegeben wird.

```
 1 { Vorabdeklaration }
 2 procedure proz2(j:real); forward;
 3
 4 { Aufrufende Prozedur }
 5 procedure proz1(i : integer);
 6     begin
 7             ...
 8             proz2(wert);
 9             ...
10     end;
```

```
11
12 { Aufgerufene Prozedur }
13 procedure proz2(i : real);
14     begin
15           ...
16     end;
```

Ihr Funktions- oder Prozedurkopf wird vollständig bekannt gemacht, allerdings wird der Prozedurkörper nicht ausgefüllt. Statt dessen wird das Schlüsselwort `forward` eingesetzt, um dem Compiler mitzuteilen, daß die Definition irgendwann später folgt. Nun kann die andere Subroutine ohne weitere Probleme Bezug auf die vorab deklarierte Subroutine nehmen.

7.2.2 Der Prozedur- und Funktionskopf

Im Kopf der Subroutine wird angegeben, von welchem Typ sie ist, wie sie heißt und welche Parameter eventuell benötigt werden. Abgeschlossen wird die Definition mit einem Semikolon. Danach folgen die Anweisungen des Prozedurkörpers.

Prozedurkopf

Ein *Prozedurkopf* beginnt mit `procedure`, gefolgt vom Namen der Prozedur. Für die Namensvergabe gelten die gleichen Regeln wie für alle anderen Bezeichner auch.

Wenn die Prozedur keine Parameter hat, ist der Kopf mit einem Semikolon abzuschließen. Sollten Parameter übergeben werden, ist die Parameterliste in runden Klammern nach dem Namen anzugeben. Erst dann erfolgt der Abschluß mit einem Semikolon.

```
procedure ErsteProzedur;
procedure ZweiteProzedur(Parameterliste);
```

Funktionskopf

Ein *Funktionskopf* ist prinzipiell gleich zum Prozedurkopf aufgebaut. Hier folgt auf das Schlüsselwort `function` der *Funktionsname.* Wenn Parameter verlangt werden, müssen sie in einer Liste innerhalb runder Klammern angegeben werden, ansonsten fällt die Klammerung weg.

Der Unterschied zur Prozedur wird auch darin auffällig, daß noch ein *Funktionstyp* für den Rückgabewert der Funktion angegeben werden muß. Dies geschieht durch die Typangabe nach einem Doppelpunkt:

```
function ErsteFunktion : Rückgabetyp;
function ZweiteFunktion(Parameterliste) : Rückgabetyp;
```

Als *Rückgabetyp* sind vordefinierte und benutzerdefinierte Datentypen erlaubt. Allerdings kann immer nur ein einziger Wert zurückgegeben werden. Daher fallen hier Typlisten, Arrays und die verschiedenen Recordtypen als Rückgabetypen aus. Um diese zu bearbeiten, muß man andere Wege gehen. Welche das sind, erfahren Sie gleich.

7.2.3 Das Innere der Subroutine

Kommen wir nun zum Innenleben einer Subroutine. Sie können alle Konstrukte einsetzen, die Ihnen bisher schon bekannt sind. Zwischen einem `begin` und `end;` stehen normale Anweisungen, Schleifen, Verzweigungen usw. Nach dem Prozedurkopf sind die folgenden Abschnitte erlaubt:

- Typvereinbarungen: `type ...`
- Konstantenvereinbarungen: `const ...`
- Variablenvereinbarungen: `var ...`
- Programmcode: `begin ... end;`

Natürlich können von hier aus auch wieder andere Funktionen und Prozeduren aufgerufen werden. In soweit besteht also kein Unterschied zu einem Programm.

```
1 procedure Begruessung;
2     var
3           name : string;
4
5     begin
6           Write('Ihr Name bitte: ');
7           ReadLn(name);
8
9           if name = 'Anne' then
10                WriteLn('Na, wie geht's?')
11          else
12                WriteLn('Guten Tag, ', name);
13    end.
```

Funktionswerte

Neu hinzugekommen ist die Variable, in der der Rückgabewert der Funktion gehalten wird. Für die Rückgabe steht Ihnen eine implizit vereinbarte Variable zur Verfügung, welche den gleichen Namen trägt wie die Funktion selber und deren Typ dem angegebenen Rückgabetyp der Funktion entspricht. Sie können diese Variable behandeln wie jede andere Variable auch, allerdings mit der Einschränkung, daß Sie sie nur beschreiben können. Sollten Sie versuchen, lesend auf sie zuzugreifen, wird das als Funktionsaufruf verstanden. Die Folge wird eine unendliche Folge von Selbstaufrufen der Funktion sein (»Rekursion«).

```
1 program Funktionstest;
2
3 function TestFunktion : integer;
4     var
5           eingabe : integer;
6     begin
7           Write('Eingabe einer Zahl: ');
8           ReadLn(eingabe);
9
10          TestFunktion := eingabe;
11    end;
```

```
12
13 var
14     i : integer;
15 begin
16     i := TestFunktion;
17     WriteLn('Ich erhielt ', i);
18 end.
```

Listing 7.2: Verwendung des Funktionsnamens als Variable (7_2.PAS)

Beim Ende der Funktion gilt der letzte, aktuell in dieser Variable stehende Wert als Rückgabewert der Funktion.

Verschachtelte Definitionen

Eine weitere Besonderheit sind verschachtelte Definitionen von Subroutinen. Das sieht so aus, daß beispielsweise innerhalb einer Prozedurdefinition weitere Prozedur- und Funktionsdefinitionen vorgenommen werden.

Diese Subroutinen sind dann aber nur innerhalb der umschließenden Prozedur zugreifbar. Von außerhalb, beispielsweise dem Hauptprogramm aus, sind sie nicht zugreifbar.

```
 1 program Verschachtelung1;
 2
 3 procedure Ebene1;
 4     procedure Ebene2;
 5             begin
 6                     WriteLn('Ebene 2');
 7             end;
 8     begin
 9             WriteLn('Ebene 1');
10             Ebene2;
11     end;
12
```

```
13 begin
14      WriteLn('Hauptebene');
15      Ebene1;
16 end.
```

Listing 7.3: Beispiel für verschachtelte Subroutinen (7_3.PAS)

Im vorangegangenen Programm haben wir zwei ineinander verschachtelte Prozeduren, nämlich `Ebene1` und darin `Ebene2` (Zeilen 3 bis 11).

Beim Aufruf des Programms wird zuerst `Hauptebene` ausgegeben, dann `Ebene 1` und zuletzt `Ebene 2`. Hätten wir versucht, `Ebene2` innerhalb der Hauptebene aufzurufen, wäre vom Compiler eine Fehlermeldung gekommen.

Sollte eventuell schon eine `Ebene2` existiert haben, wäre sie innerhalb von `Ebene1` gewissermaßen überdeckt worden. Auf der Hauptebene ist dann die ursprüngliche Prozedur wieder zugreifbar:

```
 1 program Verschachtelung2;
 2
 3 procedure Ebene2;
 4 begin
 5      WriteLn('Seltsame Sache, oder?');
 6 end;
 7
 8 procedure Ebene1;
 9      procedure Ebene2;
10              begin
11                      WriteLn('Ebene 2');
12              end;
13      begin
14              WriteLn('Ebene 1');
15              Ebene2;
16      end;
17
```

```
18 begin
19     WriteLn('Hauptebene');
20     Ebene1;
21     Ebene2;
22 end.
```

Listing 7.4: Beispiel für lokale und globale Subroutinen (🖫 7_4.PAS)

Als Ausgabe wären die folgenden Sätze erschienen:

»Hauptebene« (Zeile 19)

»Ebene1« (Zeile 14)

»Ebene2« (Zeile 11)

»Seltsame Sache, oder?« (Zeile 5)

Man kann also mit Fug und Recht davon sprechen, daß es in Pascal so etwas wie lokale und globale Subroutinen gibt. Damit sind dann recht eigentümliche Programmkonstrukte möglich.

Ähnliche Phänomene werden Sie etwas später noch bei den Datentypen, Variablen und Konstanten kennenlernen.

7.2.4 Aufruf

Der Aufruf einer selbstdefinierten Funktion oder Prozedur unterscheidet sich in nichts vom Vorgehen bei vordefinierten Subroutinen. Funktionen werden im Kontext eines Ausdrucks aufgerufen, Prozeduren stehen für sich.

Normaler Aufruf

Müssen Parameter angegeben werden, erfolgt dies innerhalb runder Klammern, die ansonsten weggelassen werden.

Die oben definierten Prozeduren und Funktionen würden dann wie folgt aufgerufen werden:

```
...
ErsteProzedur;
ZweiteProzedur( ..., ..., ... );
...
erg := ErsteFunktion;
erg := ZweiteFunktion( ..., ..., ...);
...
```

Wenn die Subroutinen Parameter erwarten, müssen diese beim Aufruf angegeben werden, und natürlich müssen sie vom Typ her mit dem übereinstimmen, was bei der Definition der Parameterliste angegeben wurde.

Rekursive Funktionen und Prozeduren

Was passiert eigentlich, wenn innerhalb eines Funktionskörpers noch einmal ein Aufruf dieser Funktion erfolgt? Der Compiler akzeptiert das bedenkenlos, und auch während der Programmausführung gibt es keinen Protest. Es handelt sich nämlich um ein Konstrukt, das vergleichsweise häufig vorkommt: *rekursive Funktionen.*

Und natürlich können auch Prozeduren rekursiv aufgerufen werden.

Ein Beispiel für eine nicht ganz glückliche Variante der Rekursion folgt hier. Das Programm läuft, bis irgendwann ein Abbruch durch das Betriebssystem erfolgt.

```
1 program Crash;
2
3 procedure Rekursiv;
4     begin
5           WriteLn('In Rekursion ');
6           Rekursiv;
7     end;
8
```

```
 9 begin
10      Rekursiv;
11 end.
```

Listing 7.5: Mißglückte Rekursion (🖫 7_5.PAS)

Um eine Rekursion sinnvoll einzusetzen, muß innerhalb der Funktion oder Prozedur ein Abbruchkriterium existieren, mit dem die maximale Anzahl der Durchläufe gesteuert wird. Das nächste Beispiel benutzt einen Parameter, der innerhalb der Rekursion immer weitergereicht wird. Dabei verringert sich sein Wert und bei 0 ist die Rekursion beendet:

```
 1 program Rekursion;
 2
 3 procedure Rekursiv(i : integer);
 4     begin
 5           if i <> 0 then begin
 6                 WriteLn('D: Stufe : ', i);
 7                 Rekursiv(i - 1);
 8                 WriteLn('U: Stufe : ', i);
 9           end;
10     end;
11
12 begin
13     WriteLn('Start');
14     Rekursiv(5);
15     WriteLn('Ende');
16 end.
```

Listing 7.6: Gelungene Rekursion (🖫 7_6.PAS)

Nun wird der Aufruf von `Rekursiv` insgesamt sechsmal ausgeführt. Ist der Wert in `i` gleich 0, bricht diese Kette ab. Und in der umgekehrten, gleichen Reihenfolge, in der das Programm in die Rekursion eingestiegen ist, taucht es aus ihr wieder auf. An den Programmausgaben können Sie es mitverfolgen.

7.3 Variablen und Subroutinen

Ein weiteres weites Feld sind die verschiedenen Variablen und Datentypen sowie ihr Einsatz innerhalb der Subroutinen. Dort unterscheidet man *Parametervariablen, lokale Variablen* und *globale Variablen.*

7.3.1 Parameter

Mit der *Parameterliste* bestimmt die Funktion, welche Werte sie erwartet, um ihre Aufgabe erfüllen zu können. Der normale Aufbau für eine Parameterangabe besteht, wie bei den Variablendeklarationen auch, aus `Parametername : Datentyp`. Eine Liste von Parametern gleichen Typs wird mit `name1, name2 : Datentyp` vereinbart. Sollen mehrere verschiedene Datentypen in der Parameterliste stehen, werden sie durch Semikola getrennt: `para1 : Typ1 ; para2 : Typ2`.

Call by Value

Häufig will man nur Daten in eine Subroutine hineinreichen, ohne daß die Originalwerte bei den nun folgenden Bearbeitungsschritten verändert werden. Diese Wertübergabe heißt *Call by Value*, ihre Definition entspricht der üblichen Art der Parameter-Vereinbarung:

```
procedure Prozedur(para : Datentyp);
```

Im folgenden Beispiel wird eine Prozedur Werte entgegennehmen, und sie verändern. Da die Werte »by Value« übergeben wurden, geschieht den Originalen nichts, wie die Ausgaben beweisen:

```
1 program CallByValue;
2
3 procedure MachWas(i : integer);
4     begin
5           WriteLn('----->', i);
6           i := 99;
7           WriteLn('----->', i);
8     end;
```

```
 9
10 var
11      i : integer;
12
13 begin
14      i := 1;
15      WriteLn(i);
16      MachWas(i);
17      WriteLn(i);
18 end.
```

Listing 7.7: Beispiel für Call by Value (🖫 7_7.PAS)

Call by Reference

Manchmal ist es wünschenswert, daß die Originale durch den Funktionsaufruf verändert werden sollen, und nicht irgendwelche Kopien. Dann reicht ein »Call by Value« nicht aus, es muß eine Referenz auf die Originalspeicherstellen benutzt werden. Hier kommt der *Call by Reference* ins Spiel:

```
procedure Prozedur(var para : Datentyp);
```

Durch das Schlüsselwort `var` vor der Parameterliste werden die nachfolgenden Parameter alle zu Referenzparametern. Die Folge ist, daß Veränderungen, die innerhalb der Subroutine ausgeführt werden, nun auch Auswirkungen auf die Originale haben. Hier sehen Sie eine Möglichkeit, wie Strukturen und Arrays ebenfalls an Subroutinen zwecks Veränderung übergeben werden können.

Das folgende Beispiel zeigt zwei Prozeduren, die scheinbar das gleiche machen: Die übergebenen Parameterwerte gegeneinander austauschen. Allerdings wird die erste Prozedur dies mit »Call by Value« und die zweite mit »Call by Reference« machen. Die Programmausgaben sprechen für sich.

```
 1 program Swapper;
 2
 3 procedure NonSwap(a, b : integer);
 4     var
 5           h : integer;
 6         begin
 7                h := a;
 8                a := b;
 9                b := h;
10         end;
11
12 procedure Swap(var a, b : integer);
13           var
14              h : integer;
15           begin
16                h := a;
17                a := b;
18                b := h;
19           end;
20
21 var
22    a, b : integer;
23
24 begin
25     a := 1;
26     b := 2;
27
28     WriteLn(a, b);
29     NonSwap(a, b);
30     WriteLn(a, b);
31     Swap(a, b);
32     WriteLn(a, b);
33 end.
```

Listing 7.8: Effekte von Call by Value und Call by Reference (🖫 7_8.PAS)

Typenlose Parameter

Beim Einsatz *typenloser Parameter* kann beim Aufruf jeder beliebige Datentyp angegeben werden. Auf der einen Seite wird der Compiler jetzt keine Typüberprüfung mehr vornehmen, auf der anderen Seite muß die Subroutine aber intern erst einmal den aktuell übergebenen Datentyp ermitteln und dann anpassen. Erst danach kann mit den darinstehenden Werten gearbeitet werden.

So kann beispielsweise die Funktion `Byte()` eingesetzt werden, um zu ermitteln, wie groß der übergebene Datentyp ist. Die nachfolgend gezeigte Funktion `Laenge()` macht davon Gebrauch:

```
 1 program Laengen;
 2
 3 function Laenge(var t) : integer;
 4     begin
 5           Laenge := Byte(t);
 6     end;
 7
 8 var
 9     i : integer;
10     s : string;
11
12 begin
13     i := 1;
14     s := 'ABCDEF';
15
16     WriteLn(Laenge(i));
17     WriteLn(Laenge(s));
18 end.
```

Listing 7.9: Beispiel für typenlose Parameter (🖫 7_9.PAS)

Die Funktion arbeitet allerdings nur mit Variablen als Parameter zusammen, da der typenlose Parameter via Call by Reference weitergegeben wird. Aber dafür gibt sie schön aus, wie viele Bytes momentan belegt sind.

Offene Parameter

Bei diesen Parametern handelt es sich um Argumente des Typs `string` oder `array`. Um offene Parameter nutzen zu können, muß allerdings die Compilerdirektive `{$P+}` eingesetzt werden.

Durch *offene Parameter* können Zeichenketten oder Arrays beliebiger Länge an ein und dieselbe Subroutine übergeben werden. Da so etwas bei Zeichenketten recht häufig vorkommt – wer weiß schon immer im voraus, wie lang ein Text wird – wurde der Datentyp *openstring* definiert. Mit ihm als Parametertyp ist es nicht einmal mehr nötig, die Compilerdirektive einzusetzen.

Das folgende Beispiel nutzt diesen Effekt aus, um unterschiedlich große Ganzzahlarrays aufzusummieren. Schauen Sie sich einmal an, was passiert, wenn die Compilerdirektive abgeschaltet ist und nur feste Arrays eingesetzt werden dürfen:

```
 1 program OffeneParameter;
 2
 3 function ArraySumme(a : array of integer) : integer;
 4 var
 5   i, sum : integer;
 6 begin
 7   sum := 0;
 8   for i := 1 to High(a) do
 9       sum := sum + a[i];
10   ArraySumme := sum;
11 end;
12
13 var
14    a1 : array [1..3] of integer;
15    a2 : array [1..10] of integer;
16    i : integer;
17
```

```
18 begin
19   { Initialisierung der Arrays mit Werten }
20   for i := 1 to 3 do a1[i] := i;
21   for i := 1 to 10 do a2[i] := i;
22
23   WriteLn(ArraySumme(a1));
24   WriteLn(ArraySumme(a2));
25 end.
```

Listing 7.10: Beispiel für offene Parameter (🖫 7_10.PAS)

Konstante Parameter

Mit dem Wort `const` können *konstante Parameter* vereinbart werden. Das hat den Vorteil, daß sie gewissermaßen mit der Option »Read only« an die Subroutine übergeben werden.

```
procedure Proz1(const a: s);
```

Obige Definition vereinbart, daß Strings übergeben, aber innerhalb der Prozedur nicht verändert werden können.

Prozedurtypen

Ohne das wir weiter darauf eingehen wollen: Es gibt in Borland Pascal eine Möglichkeit, auch die Startadressen von Prozeduren als Parameter einer anderen Funktion oder Prozedur zu übergeben!

Mit Hilfe dieser Startadresse kann die Subroutine dann die dazugehörige Prozedur starten. Das wäre noch nichts Besonderes, aber mit Hilfe der *Prozedurtypen* können beliebige, vorher vereinbarte Startadressen übergeben werden. Unendlich flexible Programme werden so möglich.

7.3.2 Globale und lokale Datenobjekte

Im Zusammenhang mit den verschachtelten Prozedur- und Funktionsdefinitionen konnten Sie schon sehen, daß Pascal bezüglich der

Subroutinen zwischen lokalen und globalen Programmobjekten unterscheidet.

Die Grundregel ist dabei immer die, daß ein extern definiertes Objekt innerhalb einer Subroutine mithin global ist. Das aber nur solange, bis ein gleichartiges Objekt gleichen Namens innerhalb der Subroutine definiert wird. Dieses lokale Objekt hat im Fall eines nun folgenden Aufrufs immer den Vorrang und überdeckt gewissermaßen das globale Objekt.

Die Überdeckung ist allerdings nur solange gültig, wie die Programmausführung sich in der Subroutine befindet, in der die Überdeckung stattfindet. Danach ist das globale Objekt wieder voll zugänglich.

Lokale Typvereinbarungen

Typvereinbarungen funktionieren genauso wie im normalen Programm. Ist ein Typ innerhalb der Subroutine definiert worden, ist er *lokal,* und von außen nicht sichtbar. So kann eine Subroutine ihre ganz privaten Typen definieren und nutzen. Eventuell vorhandene externe *(globale)* Typen gleichen Namens werden dabei durch die neue Definition überdeckt.

Im folgenden Beispiel existiert zu einem externen Typ eine lokale Variante. Innerhalb der Prozedur hat nur diese Gültigkeit, im Hauptprogramm dagegen das externe Original:

```
1 program Lokal;
2
3 type
4     TS = record
5              i : string;
6     end;
7
8 (* -- Prozedurdefinitionen -- *)
```

```
 9 procedure XYZ;
10
11 type
12     TS = record
13                 i : integer;
14          end;
15 var
16    v : TS;
17 begin
18    v.i := 99;
19    WriteLn(v.i);
20 end;
21
22 (* --  Hauptprogramm -- *)
23 var
24    v : TS;
25 begin
26       v.i := 'Hey';
27       XYZ;
28       WriteLn(v.i);
29 end.
```

Listing 7.11: Beispiel für lokale Typdefinitionen (🖫 7_11.PAS)

Globale Typvereinbarungen

Für den Fall, daß der globale Typ nicht überdeckt wird, steht er mit seiner ganzen Funktionalität auch innerhalb der Subroutine zur Verfügung:

```
1 program Global;
2
3 type
4     TS = record
5                 i : string;
6     end;
7
```

```
 8 (* -- Prozedurdefinitionen -- *)
 9 procedure XYZ;
10 var
11    v : TS;
12 begin
13    v.i := ;'99'
14    WriteLn(v.i);
15 end;
16
17 (* --  Hauptprogramm -- *)
18 var
19    v : TS;
20 begin
21      v.i := 'Hey';
22      XYZ;
23      WriteLn(v.i);
24 end.
```

Listing 7.12: Beispiel für lokale Typdefinitionen (🖫 7_12.PAS)

Globale Variablen

Ein ähnliches Phänomen wie bei den Typdefinitionen können Sie auch für Variablen erleben. Man kann hier ebenfalls zwischen lokalen und globalen Variablen unterscheiden. Und genauso wie bei den Typdefinitionen werden lokale Variablenvereinbarungen globale für den Bereich der Subroutine, in der sie definiert wurden, überdecken.

Wenn eine Variable auf der Ebene des Hauptprogramms definiert wurde, steht sie auch innerhalb der Subroutinen zur Verfügung, die in der Programmdatei stehen. Sie ist dann global.

Globale Variablen können dem Datenaustausch zwischen verschiedenen Subroutinen dienen. Allerdings ist ihre Verwendung nie ganz unproblematisch, wie das folgende Beispiel zeigt:

```
 1 program Endlos;
 2
 3 var
 4    i : integer;
 5
 6 procedure P2;
 7  begin
 8   for i:= 0 to 3 do
 9     WriteLn('In P2');
10  end;
11
12 procedure P1;
13  begin
14   for i := 0 to 4 do begin
15       WriteLn('In P1');
16       P2;
17   end;
18  end;
19
20 begin
21   P1;
22 end.
```

Listing 7.13: Unangenehme Effekte globaler Variablen (💾 7_13.PAS)

Das Programm ist formal vollkommen richtig, hat aber trotzdem den Fehler, daß es nie zu einem Ende kommt. Der Grund liegt darin, daß sowohl `P1` als auch `P2` auf die gleiche Variable `i` zugreifen. Und während die Schleife in `P2` immer nur den Wert von `0`, `1`, `2`, `3` durchläuft, passiert in `P1`, daß `i` den Wert `0`, `4`, `4` ... annimmt. Was daher kommt, daß `P2` den Inhalt von `i` ja immer wieder aufs neue nur bis `3` treibt. Damit hat `P1` keine Chance jemals ans Ende zu gelangen!

Lokale Variablen

Wenn innerhalb einer Subroutine eine Variable deklariert wird, ist sie nur für diese Subroutine und für in ihr verschachtelte Subroutinen

sichtbar. Sollte global eine gleichnamige Variable existieren, wird sie durch die lokale Deklaration innerhalb der Subroutine überdeckt. Keine andere Prozedur oder Funktion kann der Subroutine nun bei der Nutzung dieser Variablen dazwischenkommen. Mit der Anlage lokaler Variablen für `i` wäre das obige Problem gelöst.

```
 1 program Endlich;
 2
 3 var
 4    i : integer;
 5
 6 procedure P2;
 7  var
 8    i : integer;
 9  begin
10   for i:= 0 to 3 do
11     WriteLn('In P2');
12  end;
13
14 procedure P1;
15  var
16   i : integer;
17  begin
18   for i := 0 to 4 do begin
19       WriteLn('In P1');
20       P2;
21   end;
22  end;
23
24 begin
25   P1;
26 end.
```

Listing 7.14: Einsatz von lokalen Variablen (🖫 **7_14.PAS**)

Dafür hat eine lokale Variable aber andere Eigenheiten. Beispielsweise ist sie, wegen der Art wie sie verwaltet wird, immer mit einem

»Schmutzwert« gefüllt, dessen Größe niemand vorhersehen kann. Es erfolgt keine automatische Initialisierung auf 0 wie bei den globalen Variablen. Jede *Initialisierung* muß daher von Ihnen vorgenommen werden.

Weiterhin existiert eine lokale Variable nur für die Dauer, während der die Subroutine ausgeführt wird. Nach deren Ende verschwindet die lokale Variable und alles was darin stand. Dieses Problem der flüchtigen Variablen läßt sich letztendlich nur durch kluge Verwendung globaler Variablen lösen.

Das folgende Beispiel zeigt einige der Eigenheiten lokaler Variablen:

```
 1 program LokaleNebeneffekte;
 2
 3 procedure P1;
 4
 5 var
 6    i, j, k : integer;
 7 begin
 8  WriteLn(i, j, k); { Schmutzausgabe }
 9  { Initialisierung }
10  i := 0;
11  j := 1;
12  k := 2;
13  WriteLn(i, j, k); { Werteausgabe }
14  { (vergebliches) Hochzählen }
15  i := i + 1;
16  j := j + 1;
17  k := k + 1;
18 end;
19
20 begin
21  P1;
22  P1;
23 end.
```

Listing 7.15: Unangenehme Effekte lokaler Variablen (🖫 7_15.PAS)

Das Programm hat einen zweimaligen Aufruf von `P1`. Und dort wird zuerst der aktuelle (Schmutz-)Wert der drei Variablen ausgegeben. Dann werden sie initialisiert, erneut ausgegeben und nun alle um den Wert 1 hochgezählt. Leider zeigt die Ausgabe, daß das Hochzählen vergeblich war. Alle Werte sind verloren, und es steht zuerst wieder nur Speicherschmutz in den Variablen.

7.4 Die Definition von Units

Wie Sie Subroutinen aufbauen, wissen Sie nun. In vielen Fällen werden das »Einwegroutinen« sein, die nur für ein spezielles Projekt eingesetzt werden. Aber denken Sie nur einmal an das Beispiel für die Verwaltung eines Stacks, welches wir in diesem Kapitel entworfen haben.

Solch eine Aufgabe kann sicherlich öfter anfallen, so daß es sicherlich interessant wäre, eine Unit anzulegen, die solche Stackfunktionen anbietet. Und den dazu passenden Datentyp gleich dazu.

So stellt sich nun die Frage, wie eine Unit denn aufgebaut sein muß.

7.4.1 Der Aufbau einer Unit

Eine Unit ist eine Sammlung von Konstanten, Variablen, Typdeklarationen, Prozeduren und Funktionen, fast so wie bei einem Pascal-Programm. Der Unterschied zu einem Pascal-Programm besteht darin, daß die Unit nicht eigenständig lauffähig ist und erst dadurch einen Wert erhält, daß sie mit anderen Units und einem Hauptprogramm, welches die Inhalte der Unit nutzt, verbunden wird.

Gliederung

Eine Unit gliedert sich ihrerseits in mehrere Bestandteile auf. Sie beginnt mit dem Schlüsselwort `unit` und einem Namen, ähnlich dem Beginn des Hauptprogramms.

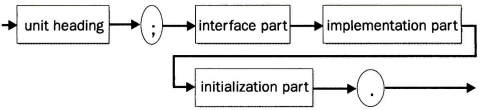


Bild 7.2: Syntaxdiagramm einer Unit

Danach folgen drei Hauptteile, nämlich der `interface`-Teil, eventuell eine eigene `uses`-Liste gefolgt vom `implementation`-Teil. Dann kann wieder eine `uses`-Liste folgen und zwischen dem letzten `begin` und `end` liegt der sogenannte Initialisierungsteil.

```
unit UnitName;

interface { Schnittstelle }

uses ...;

implementation { Definition, privater Teil }

uses ...;

begin { Beginn der Initialisierung }

end.
```

7.4.2 Der Interfaceteil

Die Schnittstelle zu den Elementen der Unit – die sie so für andere Programmeinheiten öffentlich macht – ist der sogenannte *Interfaceteil.* Hier liegen all die Variablen, Konstanten, `uses`-Klauseln, Typdeklarationen, Prozedur- und Funktionsdeklarationen, die später den Zugang zur Unit von den sie nutzenden Programmen aus ermöglichen.

Zugriff auf Units

Wenn eine Unit Bezug auf andere Units nehmen soll, werden diese zu Beginn des Interfaceteils in einer `uses`-Liste angegeben. Dabei kann es zu Problemen kommen, wenn sich zwei Units zyklisch gegenseitig aufrufen. Sollte Unit A sich hier auf Unit B beziehen, dann kann Unit B das an seiner entsprechenden Stelle nicht ebenfalls machen! Der Bezug auf Unit A muß dann im `uses`-Teil des Implementationsabschnitts von B stattfinden. Sonst sind keine weiteren Probleme zu erwarten.

Beispiel des Interfaceteils

Im Interfaceteil findet noch keine Definition der genannten Prozeduren und Funktionen statt. Es handelt sich lediglich um die Schnittstellenangabe, gewissermaßen in Form von *Prototypen.* Realisiert werden sie erst im nachfolgenden Implementationsteil.

```
unit Unit3;

interface { Hier beginnt die Schnittstelle }

uses
    WinCrt;

{ Die Unit hat eine öffentlich verfügbare Prozedur }
procedure Ausgabe(s:string);

implementation { Hier beginnt die Realisierung }
...
```

7.4.3 Der Implementationsteil

Nach der Schnittstellendefinition folgt der *Implementationsteil.* Hier werden die einzelnen Prozeduren und Funktionen der Unit realisiert.

Öffentliche und private Subroutinen

Daneben ist es möglich, auch andere Subroutinen zu definieren, die nicht öffentlich sind, und nur für den internen Gebrauch der Unit dienen sollen. Genauso ist es auch möglich, hier nur für die interne Verwendung gedachte Typen, Konstanten und Variablen zu vereinbaren. In diesem Zusammenhang können noch einmal andere Units mit einer eigenen `uses`-Klausel mit einbezogen werden.

Für die öffentlichen Funktionen und Prozeduren müssen im Interfaceteil die Parameterlisten immer angegeben werden, wenn welche gewollt sind. Für die so vereinbarten öffentlichen Subroutinen können die Parameterlisten dann im Implementationsteil weggelassen werden, was zur Ersparnis der Tipparbeit gedacht ist.

Die Parameterliste kann weggelassen werden

Das bedeutet also, daß Sie im Implementationsteil beim Funktionskopf eine im Interfaceteil angegebene Parameterliste weglassen, sie jedoch trotzdem nutzen dürfen. Ob's zur Leserlichkeit beiträgt, sei dahingestellt.

```
...
implementation { Realisierung der öffentlichen Prozedur }

procedure Ausgabe;
  begin
    WriteLn(s);
  end;
...
```

Überlagerte Units

Bei der Verwendung der mit der `uses`-Klausel eingebundenen Bibliotheken läßt sich noch eine weitere Besonderheit feststellen. Und zwar gilt, daß eine solche Bibliothek immer erst ab der Stelle nutzbar ist, die nach ihrer Einbindung mit `uses` folgt.

Außerdem wird die ursprüngliche Bedeutung einer Subroutine, die in einer eingebundenen Bibliothek existiert und nun in der aktuellen Unit noch einmal neu definiert wird, erst einmal überschrieben! Sie verschwindet allerdings nicht, sondern ist eher durch die neue Definition überdeckt. Wollen Sie die ursprüngliche Bedeutung dieser Funktion oder Prozedur trotzdem nutzen, müssen Sie bei ihrem Aufruf zusätzlich den Namen der Unit, gefolgt von einem Punkt und dem Namen der Funktion/Prozedur, angeben.

```
 1 unit Unit4;
 2
 3 interface
 4
 5 uses
 6     Unit3;
 7
 8 procedure Ausgabe(string:s)
 9
10 implementation
11
12 procedure Ausgabe;
13   begin
14     WriteLn(s);
15     Unit3.Ausgabe(s);
16   end;
17 end.
```

Listing 7.16: Beispiel für eine überlagerte Unit

7.4.4 Der Initialisierungsteil

Nach dem Implementationsteil kann noch ein Initialisierungsteil folgen. Der hier zwischen `begin` und `end` stehende Code wird immer beim Start des die Unit nutzenden Programms ausgeführt.

Einmalige Initialisierung zum Programmstart

Damit wird es möglich, die für die Verwendung der Unit vielleicht nötigen ersten Schritte automatisch vollziehen zu lassen. Sollte keine Initialisierung nötig sein, wird auf deren `begin` verzichtet und die Unit als ganzes mit einem `end.` abgeschlossen.

Das folgende Bruchstück einer Unit macht nichts sonderlich sinnvolles: Es ruft lediglich beim Programmstart die Ausgabefunktion auf und gibt seine Meldung aus.

Daran läßt sich sehr schön das Prinzip der Initialisierung erkennen: Beim Programmstart wird sie einmalig ausgeführt, danach wird der Initialisierungsteil nicht mehr durchlaufen.

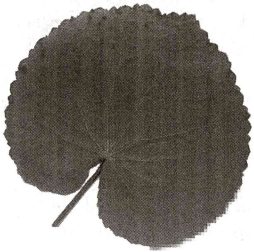
```
...
begin { Wird bei jedem Programmstart aufgerufen }
  WriteLn('Initialisierung!!!');
end.
```

Dynamische Speicherverwaltung 8

8.1 Der Hauptspeicher 222

8.2 Anlage und Zugriff 227

8.3 Zeigerinitialisierungen 233



Wenn Sie sich an die Arrays erinnern, denken Sie sicherlich vor allem an die Tatsache, daß der mit ihnen verwaltete Speicherbereich statisch ist. Ist die Anzahl der Werte bekannt, welche zu verwalten sind, macht das keine weiteren Probleme.

Aber schon bei solch einer einfachen Aufgabe, wie der einen zeilenorientierten Editor zu schreiben, werden die Arrays scheitern. Egal wie groß das Stringarray ist, es paßt nicht. Also benötigen Sie eine Möglichkeit, *dynamisch Speicher zu reservieren* und wieder freizugeben.

Daneben wünscht man sich auch schon einmal die Möglichkeit, Speicherstellen im RAM direkt anzusprechen. So etwas ist dann nötig, wenn spezielle, hardwarebezogene Programme geschrieben werden müssen.

Und zuletzt wären da noch all die ausgefeilten Programmiertechniken zu nennen, mit denen trickreiche und leistungsfähige Datenstrukturen bearbeitet werden. Dafür sind die Methoden der dynamischen Speicherverwaltung auch nicht schlecht.

Kurz und gut, es gibt eine Menge Gründe, um Ihnen in diesem Kapitel die Zeigervariablen und die pascalschen Methoden der dynamischen Speicherverwaltung zu zeigen.

8.1 Der Hauptspeicher

Um dynamischen Speicher zu verwalten, benötigt man zwei Dinge. Einmal natürlich den Speicher, und dann eine Art Aufhängepunkt, mit dem es möglich ist, den Speicher zu erreichen, um ihn zu lesen und zu schreiben.

Da die Funktionen und Operatoren der dynamischen Speicherverwaltung Adressen auf den Beginn des besorgten Speichers liefern, muß der Aufhängepunkt in der Lage sein, Adressen aufzunehmen.

Das gleiche gilt, wenn man Speicherbereich direkt ohne Umweg über Variablen ansprechen und bearbeiten will. Die Adresse des zu manipulierenden Speicherbereichs muß irgendwo festgehalten werden.

Diese Aufgaben nehmen die sogenannten *Zeigervariablen (Pointer)* wahr, die ein Datentyp speziell zur Aufnahme von Adressen sind.

8.1.1 Speicherplatz - Adresse - Variable

Die Welt der direkten Adreßzugriffe ist ganz gut zu beherrschen, wenn man sich über ihre Besonderheiten im klaren ist.

Adressen

Auf die verschiedenen Adreßschemata wollen wir hier nicht näher eingehen. Es soll lediglich daran erinnert werden, daß aus den Tagen der ersten PCs das Verfahren stammt, den Speicherraum von 1 MByte in kleinere *Segmente* von 64 KByte aufzuteilen. Um eine spezielle physikalische Adresse anzugeben, muß man also deren *Segment (Offset)* und die Adresse im Segment selber wissen. Das Schema dafür ist `Segment:Offset`, und damit wird die erste Speicherstelle im Rechner im Segment 0 und der Speicherstelle 1 liegen: `$0000:$0001`. Segmente und Offset werden in Hexadezimalzahlen angegeben, daher die Schreibweise mit dem Dollarzeichen. Um beispielsweise auf den Textbildschirm zuzugreifen, müssen Sie, je nach Grafikkarte, verschiedene Segmente angeben:

- CGA : $B800
- VGA : $A000

Diese Adreßrechnungen sind nicht unkompliziert, und sollen hier nicht weiter verfolgt werden.

Adresse - Variable

Wie hängen nun Variablen mit Zeigern zusammen? Man kann sich die Sache so vorstellen, daß der Compiler für den Programmlauf eine große Tabelle anlegt, in welcher eine Information über den Datentyp eines Programmobjekts steht, zusammen mit einem Verweis auf die Speicherstelle, wo das Objekt zu finden ist.

Damit der Zugriff für Menschen leichter wird, operiert man auf der Ebene der Programmiersprache, aber nicht etwa mit dieser Adreßinformation oder mit der Indexnummer für den dazugehörigen Tabelleneintrag, sondern mit einem Variablennamen. Wenn Sie den Namen angeben, sucht sich das Pascal-System die dazugehörige Adresse aus.

Damit steckt letztlich im Namen der Variablen unausgesprochen immer auch die Information darüber, wo sie im Speicher liegt. Und das ist eine Adresse.

Adresse - Dynamischer Speicher

Die Funktionen und Operatoren für die dynamische Speicherverwaltung arbeiten mit den Anfangsadressen der von Ihrem Programm auf dem *Heap* angeforderten Speicherblöcke. Und die Anforderungsfunktionen liefern normalerweise diese Anfangsadressen als Ergebniswerte.

Im Gegensatz zu normalen Variablen müssen Sie sich nun selber darum kümmern, daß mit dieser Adresse ein darauf verweisender Name verbunden wird. Dann ist es möglich, den Speicherbereich über diese Zeigervariable anzusprechen.

Ebenso müssen Sie sich auch um alle anderen Verwaltungsaufgaben kümmern, die sonst vom Compiler erledigt werden. Trotzdem kann sich der Aufwand lohnen, denn die größere Flexibilität des dynamischen Speichers gibt Ihnen einige interessante Möglichkeiten an die Hand.

8.1.2 Stack- und Heapgrößen

Bleibt noch zu klären, wie der dynamische Speicher organisiert wird.

Code - Daten - Heap - Stack

Programme sind, wenn sie ablaufen, in mehrere Bereiche aufgeteilt. Manche davon sind statisch, andere verändern ihre Grenzen dynamisch. Statisch sind der Bereich für den ausführbaren Code

(Codesegment) und für die schon zur Übersetzungszeit feststehenden Daten *(Datensegment)*. Daneben gibt es noch zwei dynamische Bereiche, nämlich den *Heap* und den darüberliegenden *Stack*. Die unterste Grenze des Heap liegt bei einer tiefen Speicheradresse, die unterste Grenze des Stack bei einer hohen Speicheradresse. Beide wachsen und schrumpfen im Programmverlauf, wobei die Wachstumsrichtungen immer aufeinander zuweisen.

Während globale Variablen ihren Speicherplatz aus dem Datensegment zugewiesen bekommen, erhalten lokale Variablen ihre auf dem sogenannten Laufzeitstack. Das ist auch der Fall, wenn Sie Parametervariablen anlegen. Sollten Sie zuviel Platz verbrauchen, erhalten Sie einen Laufzeitfehler.

Wenn Sie dynamischen Speicher anfordern, wird dieser auf dem Heap besorgt, dessen Ausdehnung dann entsprechend anwächst. Auch hier gibt es Grenzen der maximalen Größe. Der Versuch, sie zu überschreiten, führt aber nicht zu einem Laufzeitfehler, sondern dazu, daß die Funktion, welche den Speicherplatz besorgen soll, einen Fehlerwert zurückgibt. Ignoriert man den, kann es zu ziemlich chaotischen Programmabläufen kommen.

Stack- und Heapwerte

Die Größe von Heap und Stack kann zur Übersetzungszeit in gewissen Grenzen angegeben werden. Dies geht entweder über Compilerdirektiven oder durch die dazugehörigen Dialoge des Compilers. Vorgegeben sind die folgenden Werte:

Tabelle 8.1: Stackwerte

Speichermodell	Standard	Maximum	Minimum
Real	16384	65520	1024
Protected	16384	65520	1024
Windows	8192	65520	1024

Für die Heapwerte müssen verschiedene Plattformen unterschieden werden.

Tabelle 8.2: Zielplattform: Windows

Einstellung	Werte
Standard:	8192 Byte
Maximum:	655360 Byte

Tabelle 8.3: Zielplattform: Real, Protected Mode

	Standard	Minimum	Maximum
Min.	0	0	655360
Max.	655360	0	655360

8.1.3 Speicherprobleme

Natürlich ist der direkte Umgang mit den Ressourcen eines Rechners immer auch mit gewissen Risiken verbunden. Dies liegt weniger daran, daß die Befehle besonders gefährlich wären. Meist ist es die Gedankenlosigkeit des Programmierers, welche zu Ärger führen kann.

Illegale Speicherzugriffe

Fehler wirken sich im Umgang mit Zeigern oder dynamischem Speicher sehr viel stärker aus als beispielsweise normale Variablenfehler. Immerhin gibt es keinen Schutzmechanismus von Pascal, der verhindert, daß Sie unter MS-DOS mit Hilfe einer Zeigervariablen das Betriebssystem modifizieren. Meistens war's das dann auch.

Unter Windows sieht die Sache so aus, daß das Betriebssystem versucht, sich zu schützen. Wenn es einen illegalen Speicherzugriff bemerkt, wird die Anwendung abgeschlossen. Im darauf folgenden Fehlerdialog – leider nur bei Windows 95 – können Sie den Registerinformationen ES und EDI entnehmen, mit welchem Adreßwert eine illegale Operation versucht wurde.

Man sollte Zeigervariablen immer mit Bedacht und vor allem nach einer Initialisierung auf gültige Speicherbereiche benutzen. Während

globale Variablen, und damit auch globale Zeigervariablen, immer auf 0 initialisiert sind, müssen Sie bei lokalen Variablen aufpassen, da hier Speichermüll in der Zeigervariablen steht, den man besser nicht als Adresse interpretieren sollte.

Verlorener Speicher

Ein anderes Problem ist der verlorene Speicher. Wenn ein Programm sich Platz vom Heap geholt hat, muß es sich um dessen gesamte Verwaltung selber kümmern. Angenommen, der dafür zuständige Zeiger wird versehentlich an einen anderen Speicherbereich gehängt. Dann haben Sie die Adresse des ursprünglichen Speichers verloren. Und die ist nicht wiederzuholen! Der damit zusammenhängende Speicher bleibt bis zum Programmende gesperrt. Sollte so etwas öfter passieren, kann es sein, daß das Programm wegen Speichermangels stehen bleibt, obwohl bei richtiger Speicherverwaltung noch genügend Platz gewesen wäre! Sie müssen sich also tatsächlich selber darum kümmern, daß nicht mehr benötigter, dynamischer Speicher auch freigegeben wird.

8.2 Anlage und Zugriff

Die Anlage einer Zeigervariablen ist unproblematisch. Es wird mit dem Circonflex-Zeichen (^) als Symbol für »Zeiger auf« gearbeitet. Damit vereinbaren Sie typisierte Zeiger. Daneben gibt es noch einen Spezialfall der untypisierten Zeiger und den vorbereiteten Zeigertyp `PChar`.

8.2.1 Anlage der Zeiger

Pascal unterscheidet zwischen *typisierten* und *nicht typisierten* Zeigern. Bei typisierten Zeigern wird mit angegeben, wie die Speicherbereiche gegliedert sind, auf die zugegriffen wird.

Typisierte Zeiger

Erinnern Sie sich nur an die Diskussion der Datentypen. Dort haben Sie erfahren, daß alle Datentypen bestimmte Größen im RAM benötigen. Ein `char` gibt sich mit 1 Byte zufrieden, ein `integer` benötigt dagegen 2 Byte. Und eine `real`-Variable 6 Byte.

Wenn nun ein typisierter Zeiger angelegt wird, gibt man einen vordefinierten oder benutzerdefinierten Datentyp mit an. Somit weiß der Compiler, welche Art von Daten später über den Zeiger hinweg genutzt werden sollen und wie der Platzbedarf, die erlaubten Operationen und die Art der Speicherung sind.

Besonders wichtig sind typisierte Zeiger, wenn Sie im Rahmen dynamischer Speicherverwaltung größere Speicherblöcke anlegen lassen. Durch die Typangabe kennt der Compiler nun die Schrittweite, um von der ersten Speicheradresse zu allen weiteren zu gelangen.

Nicht typisierte Zeiger

Nicht typisierte Zeigervariablen können ins Spiel kommen, wenn man sich Strukturen schaffen will, an die beliebige Datentypen angehängt werden können. Während typisierte Zeiger gewissermaßen Spezialhaken sind, entsprechen nicht typisierte Zeiger Allzweckhaken.

Bevor auf die Inhalte, auf die ein nicht typisierter Zeiger verweist, zugegriffen werden kann, muß man allerdings einige Vorbereitungen treffen. Denn natürlich weiß der Compiler bei diesen Zeigern nicht, welche Schrittweite und Speicherform benötigt werden.

8.2.2 Anlage einer typisierten Zeigervariablen

Eine typisierte Zeigervariable legen Sie ähnlich einer normalen Variablen an. Lediglich die darauf folgende Verwendung muß erst einmal vorbereitet werden, da der frisch angelegte Zeiger normalerweise noch nicht auf gültige Speicherbereiche zeigt.

Variablendefinition

Zur Anlage geben Sie zuerst die Namensliste an, gefolgt von einem Doppelpunkt. Danach kommen der Circonflex und der Datentyp:

```
var
    zeigername : ^Datentyp;
```

Namenskonventionen

Die Namensvergabe ist, wie bei allen Variablen, beliebig. Trotzdem sollten Sie sich angewöhnen, Zeigervariablen durch einen auffälligen Namen kenntlich zu machen. Wir schlagen vor, daß Sie ein Namenspräfix als Datentyp und das Kürzel `p` für »Pointer« benutzen:

```
var
    cpwert : ^char;
    ipwert : ^integer;
    rpwert : ^real;
```

Zeigerdatentypen

Natürlich können Sie auch eigene *Zeigerdatentypen* vereinbaren, von denen wiederum Variablen abgeleitet werden. Es sind Arrays von Zeigern möglich, und als Parameter von Subroutinen hat man sie auch schon gesehen. Besonders beliebt ist ihr Einsatz im Zusammenhang mit Strukturen.

```
type
    TCharZeiger = ^char;
    TIntZeiger = ^integer;
    TRealZeiger = ^real;

var
    cpwert : TCharZeiger;
    cparray : array [1..10] of TIntZeiger;
```

Ein so angelegter Zeiger kann nun genutzt werden; allerdings sollten Sie berücksichtigen, daß er noch nicht auf einen gültigen Speicherbereich zeigt. Noch ist es ein Haken, an dem nichts hängt.

Oder Sie definieren einen Arraytyp und können nun darauf einen Zeiger vereinbaren. Lassen Sie sich nicht durcheinander bringen: Ein Zeiger auf ein Array ist etwas vollkommen anderes als ein Array von Zeigern!

```
type
    TIntArray = array [1..10] of integer;

var
    ap : ^TIntArray;
```

Zeigerketten

Zeiger sind ganz normale Datentypen. So wie mit diesen typisierte Zeiger vereinbart werden können, ist es auch erlaubt, typisierte *Zeiger auf Zeiger* zu realisieren. Und für diese *Doppelpointer* gelten die gleichen Regeln wie für normale Zeigervariablen auch.

Wie solch ein Doppelpointer realisiert werden kann, zeigt das folgende Beispiel:

```
type
    TStringPointer = ^string;

var
    sppanker : ^TStringPointer;
```

Mit *Doppelpointern* lassen sich sehr schön dynamische mehrdimensionale Arrays (Matrix) realisieren. Und wer sagt denn, daß man auf Doppelpointer beschränkt sein muß?

8.2.3 Andere Zeigertypen

Neben den typisierten Zeigern können Sie noch die schon erwähnten nichttypisierten Zeiger und einen Zeiger für den Zugriff auf Zeichenketten im C/C++-Stil nutzen.

Nicht typisierte Zeigervariablen

Für die Benutzung nicht typisierter Zeiger wurde der Datentyp Pointer angelegt. Die Vereinbarung einer solchen Variablen ist dann wie üblich auszuführen. Auch hier empfehlen wir wieder einen auffälligen Namen, wir bevorzugen dafür ein dem Namen vorangestelltes `p`:

```
var
    pvariable : pointer;
```

Der nichttypisierte Zeiger kann nun genutzt werden, allerdings gilt auch hier, daß erst einmal ein gültiger Speicherbereich mit ihm verbunden werden muß.

Zugriff auf String im C/C++-Stil: PChar

Ein spezieller Zeiger wurde von Borland vorbereitet, damit Pascal auch mit Zeichenketten im Stil von C/C++ zurechtkommt. Dies ist für die Windows-Programmierung sehr wichtig.

Der Datentyp `PChar` ist aber nichts weiter als ein Zeiger auf `char`-Bereiche. Die Datentypdefinition ist also:

```
type
    PChar = ^char;
```

Für seine Benutzung gelten die gleichen Regeln wie für alle anderen Zeiger auch.

8.2.4 Zugriff auf Zeigerinhalte

Der Zugriff auf Zeiger setzt natürlich voraus, daß sie mit gültigen Speicherbereichen verbunden sind. Als Zugriffsoperator dient das Circonflex-Zeichen.

Zugriffe

Der Inhalt einer Zeigervariablen ist nicht lesbar. Sie können allerdings mit einer normalen Zuweisung einen Adreßwert an den Zeiger binden:

```
zeiger := Adresswert;
```

Um auf den Inhalt der Speicherstelle zuzugreifen, auf die der Zeiger verweist, müssen Sie das Circonflexzeichen dem Namen des Zeigers nachstellen. Solch ein Zugriff kann sowohl lesend als auch schreibend sein.

```
{ Schreibender Zugriff }
zeiger^ := Wert;

{ Lesender Zugriff }
variable := zeiger^;
```

Das funktioniert auch ganz gut bei Arrays von Zweigen oder Zeigern, die auf Arrays verweisen.

```
{ Zeiger, der auf Array verweist }
zeiger^[i] := Wert;
variable := zeiger^[i];

{ Array von Zeigern }
array[i]^ := Wert;
variable := array[i]^;
```

Vergleiche

Ansonsten sind nur noch Vergleiche auf Gleichheit oder Ungleichheit der Speicheradressen in zwei Zeigern erlaubt. Alle anderen Operationen sind dagegen für Zeiger nicht verfügbar.

8.3 Zeigerinitialisierungen

Nachdem die Zeiger angelegt wurden, müssen sie mit gültigen Speicherbereichen verbunden werden. So ist es beispielsweise möglich, ein Array, eine einzelne Variable oder eine feste Adresse im RAM mit einem Zeiger zu verbinden.

8.3.1 Initialisierung mit festen Datenobjekten

Die einfachste Form der Initialisierung bezieht sich auf feste Datenobjekte. Damit meinen wir sowohl die Kennung für leere Zeiger als auch den Zugriff auf schon definierte Variablen beliebiger Datentypen und natürlich den Zugriff auf fixe *Speicheradressen* im PC.

Adressen ermitteln

Um die Adresse einer Variablen zu ermitteln, kann entweder der Operator @ benutzt werden, oder die Funktion `Addr()`. In jedem Fall erhalten Sie so die Anfangsadresse des angegeben Datenobjekts zurück.

```
var
    ipa, ipb : ^integer;
    i : integer;

begin
    ipa := @i;
    ipb := Addr(i);
    ...
```

Um ein Array so zu verwalten, gehen Sie den folgenden Weg. Lassen Sie sich von der eigentümlichen Schreibweise der Zugriffe dieses Zeigers auf Arrays nicht verblüffen. Im nächsten Abschnitt werden wir noch näher darauf eingehen:

```
 1 program ArrayZeiger;
 2
 3 type
 4     TIntArray10 = array [1..10] of integer;
 5
 6 var
 7     a : TIntArray10;
 8     ap : ^TIntArray10;
 9
10 begin
11     ap := @a;
12
13     ap^[1] := 1000;
14     ap^[2] := 1001;
15
16     WriteLn(ap^[1], ap^[2]);
17     WriteLn(a[1], a[2]);
18 end.
```

Listing 8.1: Beispiel für Array und Zeiger (8_1.PAS)

Angabe fester Adreßbereiche

Um eine Adresse aus Segment und Offset zu erzeugen, steht die Funktion `Ptr()` zur Verfügung. Sie geben zuerst das Segment und dann den Offset innerhalb des Segments an. Daraus wird eine reale Speicheradresse erzeugt, die dann angesprochen werden kann. Das folgende Bruchstück macht genau dies, um den Bildschirmspeicher »in den Griff« zu bekommen:

```
var
    cpscreen : ^char;

begin
    cpscreen := Ptr($b800, $0000);
    ...
```

Der Wert nil

Der Wert `nil` ist ein sogenannter »Nullpointer«. Mit `nil` als Zuweisungswert wird kenntlich gemacht, daß die Zeigervariable momentan nicht auf gültigen Speicher zeigt. Statt sie nun zufällig in Speicherbereiche zeigen zu lassen, gibt man ihr lieber einen definierten Wert.

Der Wert von `nil` entspricht im Grunde der Speicheradresse `$0000:$0000`. Das Pascal-System läßt zu, daß `nil` als nicht typisiert betrachtet und daher jeder Zeigervariablen beliebigen Typs zugewiesen werden kann.

Sie können den `nil`-Wert sowohl durch direkten Vergleich als auch mit Hilfe der Funktion `Assigned()` testen. Letztere liefert den Wert `true` zurück, wenn ein Zeiger mit einer Adresse verbunden ist, ansonsten den Wert `false`, wenn `nil` in ihm steht:

```
 1 program ZeigerTest;
 2
 3 var
 4    i : integer;
 5    ip : ^integer;
 6
 7 begin
 8      WriteLn(assigned(ip), ' ', ip = nil);
 9      ip := @i;
10      WriteLn(assigned(ip), ' ', ip = nil);
11      ip := nil;
12      WriteLn(assigned(ip), ' ', ip = nil);
13 end.
```

Listing 8.2: Die Verwendung von nil (🖫 8_2.PAS)

8.3.2 Dynamische Speicherverwaltung

Die *dynamische Speicherverwaltung* kennt unter Borland Pascal zwei Möglichkeiten, den gewünschten Platz auf dem Heap zu reservieren. Das klassische Verfahren benutzt die Prozeduren `New()` und `Dispose()`, um Platz für Datenobjekte anzulegen bzw. wieder freizugeben.

Da mit diesen Verfahren gewisse Einschränkungen verbunden sind, hat Borland die Funktionen `GetMem()` und `FreeMem()` eingeführt, mit denen die Belegung des Speichers flexibler erfolgen kann.

Die Prozeduren New() und Dispose()

Mit `New()` wird ein neues Datenobjekt angelegt und mit dem Zeiger verbunden. Die Information über den Aufbau des Datenobjekts holt sich `New()` indirekt aus dem typisierten Zeiger.

```
type
    TIntArray100 = array[1..100] of integer;

var
    ap : ^TIntArray100;

begin
    New(ap);

    ap^[1] := 1000;
    WriteLn(ap^[1]);
    ...
```

Mit dem so angelegten Datenobjekt kann nun so umgegangen werden, wie mit jedem Zeiger auf eines der vorher beschriebenen festen Datenobjekte. Doch Vorsicht! Sie können nach wie vor ein beliebiges anderes Objekt gleichen Typs an den nun eigentlich vergebenen Zeiger hängen. Wenn Sie das machen, verlieren Sie die Adresse auf das Datenobjekt. Leider wird es dadurch nicht automatisch freigegeben, sondern liegt jetzt gewissermaßen als Speicherleiche auf dem Heap.

Um ein Datenobjekt, welches mit `New()` angelegt wurde, wieder freizugeben, müssen Sie `Dispose()` aufrufen. Wieder wird der Zeiger als Argument übergeben, den Rest erledigt dann das Pascal System:

```
...
    Dispose(ap);
end.
```

Es ist also sicherlich eine gute Idee, alle nicht mehr benötigten Datenobjekte sofort wieder freizugeben. Wenn das Programm endet, wird sein Heap abgebaut, und damit alle noch verbliebenen Objekte, ob gültig oder Speicherleiche, beseitigt.

Die Prozeduren GetMem() und FreeMem()

Das Verfahren, mit `New()` Speicherplatz zu belegen, ist sicherlich nicht schlecht. Aber leider ist es manchmal zu unflexibel. Statt eines Datenobjekts muß es eine ganze Gruppe davon sein. Mit `New()` ist man nun gezwungen, Arrays und ähnliche Konstrukte zur Hilfe zu nehmen.

Oder Sie benutzten `GetMem()`. Die Prozedur erwartet einen beliebigen Zeiger und die Information, wieviel Speicher es denn dafür sein darf. Nachteil des Verfahrens ist der, daß Sie diese beliebige Größe in Byte angeben müssen. Und wie groß ist denn nun ein `real`-Datentyp? Oder Ihre Struktur? Mit `SizeOf()` steht eine Funktion zur Verfügung, welche die Größe eines Datenobjekts in Byte zurückliefert.

Damit würde der optimale Aufruf von `GetMem()` dann aussehen wie das folgende Beispiel:

```
GetMem(P, SizeOf(Datentyp));
```

Um den Speicher wieder freizugeben, müssen Sie die Prozedur `FreeMem()` einsetzen. Bei ihrem Aufruf geben Sie den Zeiger und die freizugebende Anzahl von Bytes an. Dieser Wert muß exakt identisch sein zu dem, der bei `GetMem()` eingesetzt wurde.

Es ist zwar möglich, `New()` zusammen mit `FreeMem()` und `GetMem()` zusammen mit `Dispose()` zu benutzen, ob das aber wirklich immer problemlos bleibt, vermögen wir nicht zu sagen, und so raten wir von der gemischten Benutzung dieser Prozeduren ab.

Gelöschte Zeiger kennzeichnen

Sollten Sie übrigens einmal nach einem Aufruf von `Dispose()` oder `FreeMem()` einen Test mittels `Assigned()` ausführen, ob automatisch ein `nil` gesetzt wurde, werden Sie etwas Erstaunliches erleben: `Assigned()` liefert nämlich `true` zurück.

Scheinbar ist also die Speicheradresse des Objekts noch mit dem Zeiger verbunden! Wenn Sie den so scheinbar noch vorhandenen Speicher nutzen wollen, erleben Sie allerdings eine böse Überraschung.

```
program Unsinn;

var
    ip : ^integer;

begin
    WriteLn(Assigned(ip));
    New(ip);
    WriteLn(Assigned(ip));

    ip^ := 99;
    WriteLn(ip^);

    Dispose(ip);
    WriteLn(Assigned(ip));

    { jetzt gibt's einen Crash! }
    { WriteLn(ip^); }
end.
```

Listing 8.3: Was beim Zugriff auf ungültige Zeiger passiert (8_3.PAS)

Normalerweise wird das Programm als Ausgaben FALSE, TRUE, TRUE produzieren. Wenn Sie allerdings den Kommentar um das letzte `WriteLn()` entfernen, wird es mit einem Laufzeitfehler abbrechen. Das passiert, wenn ein nicht mehr gültiger Zeiger benutzt wird.

Normalerweise sollte man einen Speicherzugriff mit Zeigern immer davon abhängig machen, ob der Zeiger gültig ist, was ja mit `Assigned()` ganz gut getestet werden kann. Damit dieser Test wirklich immer anwendbar ist, empfiehlt es sich, daß Sie nach einer Zeigerfreigabe eigenhändig ein `nil` dort eintragen.

```
program SichererZugriff;

var
   ip : ^integer;

begin
   New(ip);

   if Assigned(ip) then begin
        ip^ := 1000;
        WriteLn(ip^);
   end else
        WriteLn('Zeigerfehler!');

   Dispose(ip);
   ip := nil;

   if Assigned(ip) then
        WriteLn(ip^)
   else
        WriteLn('Zeigerfehler!');
end.
```

Listing 8.4: Sicherer Zeigerzugriff (🖫 **8_4.PAS**)

8.4 Zum Abschied

Damit sind wir am Ende dieses Pascal-Kurses angekommen. Sie sollten nun genug Grundlagenwissen erworben haben, um weitere interessante Pascal-Literatur »verschlingen« zu können – hoffen wir.

Wo sollten Sie weitermachen? Es bietet sich an, nun einige Dinge über Datentypen und Datenstrukturen in Erfahrung zu bringen. Dazu gibt es eine riesige Menge an Literatur. Unser Favorit ist das Buch »Algorithmen« von R. Sedgewick aus dem Addison-Wesley Verlag.

Danach wissen Sie genug über die Programmierung, um mit den objektorientierten Erweiterungen von Borland-Pascal zu beginnen. Und sicherlich werden Sie sich zu dem Zeitpunkt auch schon einmal mit Delphi befaßt haben, einer Programmierumgebung für Windows, die auf Pascal und Object Pascal basiert. Versuchen Sie es, es macht wirklich Spaß! Und lassen Sie sich vor allem nicht entmutigen, wenn es einmal nicht so klappt. Sie dürfen nur nicht aufgeben; Sturheit ist neben geistiger Flexibilität eine der hervorstechenden Eigenschaften guter Programmierer.

Viel Spaß!

Pascal im Überblick A

A.1 Compiler-Direktiven 242

A.2 Parameterbefehle 242

A.3 Befehle zur bedingten Compilierung..... 243

A.4 Reservierte Worte..... 243

A.5 Standardanweisungen..... 245

A.1 Compiler-Direktiven

Tabelle A.1: Compiler-Direktiven

Compiler-Direktive	Bedeutung
$A	Ausrichtung der Variablen
$B	Auswertung Boolscher Ausdrücke
$D	Debugger-Information
$F	Erzwingen von FAR-Aufrufen
$G	Codeerzeugung für 80286
$I	Automatische Prüfung von Ein-/Ausgaben
$K	Smart Callbacks
$L	Lokale Symbole
$N	Numerische Operationen des 80x87
$P	Offene Array-Grenzen
$R	Bereichsüberprüfung
$S	Stack-Prüfung
$T	Typzeiger-Prüfung
$V	Überprüfung von Var-Strings
$W	Windows Stack Frame
$X	Erweiterte Syntax
$Y	Symbol-Information

A.2 Parameterbefehle

Tabelle A.2: Parameterbefehle

Parameterbefehl	Bedeutung
$Cxxx	Code-Segment-Attribute
$Dxxx	Beschreibung
$Ixxx	Include-Datei
$Lxxx	Einbinden von Object-Dateien
$Mxxx	Speicherzuweisungsgröße
$Rxxx	Ressourcen-Datei
$Sxxx	Segmentgröße

A.3 Befehle zur bedingten Compilierung

Tabelle A.3: Befehle zur bedingten Compilierung

Befehl	Bedeutung
DEFINE	Definition eines Makros
ELSE	Alternative der Steueranweisung
ENDIF	Ende der Steueranweisung
IFDEF	Wenn ein Makro definiert wurde
IFNDEF	Wenn ein Makro nicht definiert wurde
UNDEF	Makrodefinition löschen

A.4 Reservierte Worte

Tabelle A.4: Reservierte Worte

Wort	Bedeutung/Bereich
and	Logischer Operator (UND)
asm	Anweisungen in Maschinensprache
array	Datenfeld
begin	Blockbeginn
case	Mehrfachauswahl
const	Konstantendefinition
constructor	Konstruktor eines Objekts
destructor	Destruktor eines Objekts
div	Division
do	Schleife
downto	Schleifenzähler
else	Verzweigung
end	Blockende
exports	DLL-Funktionen
file	Dateityp
for	Zählschleife
function	Funktionsdefinition
goto	Sprung

Tabelle A.4: Reservierte Worte (Fortsetzung)

Wort	Bedeutung/Bereich
if	Verzweigung
implementation	Modul-/Unitdefinition
in	Mengenoperator
inherited	Vererbungsangabe für Objekte
inline	Maschinensprachebefehle
interface	Schnittstelle eines Moduls
label	Definition von Sprungzielen
library	DLL-Definition
mod	Divisionsrest
nil	Null-Zeiger
not	Logischer Operator (NICHT)
object	Objektdefinition
of	Typdefinition
or	Logischer Operator (ODER)
packed	Speicherorganisation von Datentypen
procedure	Prozedurdefinition
program	Programmkopf
record	Benutzerdefinierter Datentyp
repeat	Schleife
set	Mengendatentyp
shl	Bitoperator
shr	Bitoperator
string	Zeichenkette
then	Verzweigung
to	Zählschleife
type	Typdefinition
unit	Modulkopf
until	Schleife
uses	Modulzugriff
var	Variablendefinition
while	Schleife
with	Zugriff auf Records
xor	Logischer Operator (EXCLUSIV-ODER)

A.5 Standardanweisungen

Tabelle A.5: Standardanweisungen

Wort	Bedeutung/Bereich
absolute	Variable an einer absoluten Adresse
assembler	Anweisungen in Maschinensprache
export	DLL-Zugriff
external	Modulübergreifende Variablen
far	Adressierungsart
forward	Vorabdeklaration einer Prozedurdefiniton
index	DLL-Zugriff
interrupt	Deklaration von Interrupt-Prozeduren
name	DLL-Zugriff
near	Adressierungsart
private	Objektdefinition
public	Objektdefinition
resident	DLL-Zugriff
virtual	Objektmethodendefinition

Die Diskette B

Auf der beiliegenden Diskette finden Sie – nach Kapiteln geordnet jeweils in einem eigenen Verzeichnis – die Programm-Listings aus dem Buch.

Zusätzlich zu diesen Beispielen enthält die Diskette eine Auswahl von Shareware-Programmen, Source-Codes und Tools zum Themenbereich Pascal. Insgesamt handelt es sich dabei um mehr als 3 MByte an Daten, in denen Sie nach Belieben stöbern und dadurch weitere Pascal-Programmiertechniken erlernen können.

Aus Platzgründen liegen diese Zusatz-Tools in komprimierter Form in der Datei *BIB.ARJ* vor. Mit dem Dekomprimierungsprogramm *ARJ.EXE,* das sich ebenfalls auf der Diskette befindet, können Sie das Archiv expandieren. Nehmen Sie dazu folgende Schritte vor:

- Erstellen Sie auf Ihrer Festplatte ein neues Verzeichnis, und nennen Sie es beispielsweise *PASTOOLS.*
- Kopieren Sie die Dateien *ARJ.EXE* und *BIB.ARJ* in dieses Verzeichnis.
- Wechseln Sie in das neu angelegte Verzeichnis.
- Geben Sie dort folgenden Befehl ein:
  `arj x bib.arj` ↵
- Das Archiv-Programm fragt Sie, ob es das Verzeichnis neu erstellen soll. Geben Sie A ↵ ein (für always), damit alle Verzeichnisse automatisch neu erstellt werden.

Index

!

- 74
'\0' 65
( ) 74
+ 74
< 74
<> 74
= 74
=< 74
> 74
>= 74
@ 74; 233
^ 227

A

Addr() 233
Adressen, ermitteln 233
Alphabet 23
Alternativen e. varianten Record . 157
and 74
Anweisung 72
Anweisungsblock 72; 75
Argumente 103
Array 45; 145; 146
  Dimensionsangabe 147
  Index 148
  Indexgrenzen 147
  in Strukturen 156
  mehrdimensionales 147
Arrayelemente, Zugriff auf 148
ASCII
  Alphabet 44
  Konvertierung 57
Assign 169
Assign() 164
Assigned 235
Aufruf, einer Funktion
  oder Prozedur 200
Aufzählungen 129; 137
Ausdruck 72
  logischer 47
  Prioritäten 74
  Seiteneffekte 73
  Verschachtelung 73
Ausdruckswerte 72
Ausgabe, formatierte 36

B

begin 30
Bezeichner, Namensregeln 25
Bibliotheken 21; 189
  Dateien 21
  einbinden 189
  selbstdefinierte 190
  vordefinierte 190
Bitshift-Operatoren 88
boolean 62; 80
Boolsche Zahlen, Regeln 62
Break 121; 123
byte 61; 167
Byte() 206
bytebool 62

C

Call by Reference 204
Call by Value 203
case 109; 157
char 52; 92

Chr() 57
Close 169
Close() 164
Codesegment 225
COM 21
comp 63
Compilerdirektiven 27
Compilierung 22
Concat 67
const 49; 135
Continue 121; 123
Copy 67
Crt 190

D

Datei
  Fehlerbehandlung 173
  Grundoperationen 162
  lesen 172
  lesen und schreiben 169
  öffnen 169
  schließen 173
  schreiben 172
  Schritte der Bearbeitung 165
  typisierte 165
  untypisierte 168
Dateifunktionen 169
Dateitypen 44
Dateivariable 162
  Deklaration 165
  für Textdateien 163
Dateizugriff 162
Datenobjekte
  globale und lokale 208
Datensegment 225
Datentypen 40
  benutzerdefinierte 45
  Eigenschaften ordinaler 54
  für Arrays 147
  ordinale 42
  reale 43
  vordefinierte 42
Definition einer symbolischen
  Konstanten 49
Delete 67
Dezimalsystem 47
Dispose() 236
div 4; 77
Division 77
Doppelpointer 230
Dos 190
double 63
downto 118
Dynamische Speicherverwaltung 236

E

Ein- und Ausgabekanäle 30
Eingabefunktionen 68
Einzelschritt 96
else 106; 110
end 30
Endlosschleifen 117
Eof 169
Eof(datei) 164
Exclude 144
EXE 20
Exit 121
Exponentialformat 36
extended 63

F

False 43; 47
Fehlerbehandlung
  bei Dateizugriffen 173
Felder 45; 129
File of 165
FileMode 171
FilePos() 173
Fließkommakonstante 36
Fließkommatypen 63
for.. 118
for-Schleife 118
forward 195
Forward-Deklaration 194
FreeMem() 236
function 196
Funktionen 21; 102; 187
  Rückgabetyp 196
Funktionskopf 196
Funktionsname 196
Funktionstyp 196
Funktionswerte 197

G

Ganzzahldarstellung 60
  negative 60
Ganzzahltypen, Erweiterung 61
GetMem() 236
Gliederung einer Unit 215
goto 121
goto Name 121
Graph 190
Graph3 190

H

Halt 121; 125
Heap 224; 225
Heapwerte 225
Hexadezimalzahl 47
High 55

I

if 104
if...then 105
Implementation 216
Implementationsteil 217
in 74; 142; 144
Include 144
Indexgrenzen 147
Informationsverluste
  durch Typanpassungen 92
Inhalt einer Zeigervariablen 232
Initialisierung lokaler Variablen ... 214
Initialisierungsteil 219
Insert 67
Int 92
integer 52; 58; 61
interface 216

K

Klassen 46
Kommentarblock 28
Kommentare 27
Komplement, Bitoperator 86
Konkatenation
  von Zeichenketten 67
Konstante 46
  literale 46
  numerische 36
  symbolische 48

L

Laufvariable 115
Laufzeitbibliothek 21
Length 67
Logische (Boolsche) Operatoren 81
longbool 62
longint 61
Low 55

M

Matrix 147
Mehrfachauswahl 99-100; 109
Menge
  Abfrage 142
  leere 142
  Operatoren und Funktionen .. 143
Mengenkonstruktoren 141
Mengentyp 140
mod 74; 77
Modulo 77

N

Nachkommastellen 37
  Angabe 38
Namenskonventionen 26
  bei Zeigern 229
  für Typvereinbarungen 131
Namensregeln für Bezeichner 25
Nassi-Shneiderman 96
New() 236
nil 235
not 74; 82
Nullpointer 235

O

ODER bitweise 86
of 110
Offset 223
Operanden 72
Operatoren 72; 76
  arithmetische 76
  bitweise 84
  Reihenfolge 74
or.. 74
Ord 55; 92
Ordinalroutinen 55
Ordinalzahlen 43
Overlay 190

P

PAP 97
Parameter 103; 203
  konstante 208
  offene 207
  typenlose 206
Parameterliste 203
  in Units 218
Parametername, Datentyp 203
Pascal, Turbo- 20
PChar 65; 227
Pointer 45
Pos 67
Pred 55
Printer 190
procedure 195
program 20; 30
Programmablaufpläne 97
Programmaufbau 29
Projekte 192
Protected-Mode 191
Prototypen 217
Prozeduren 21; 102; 187
Prozedurkopf 195
Prozedurtypen 208
Ptr 92
Ptr() 234

Q

Quelldateien 22

R

Read 169
Read() 69
ReadLn() 69
real 44; 52; 63
Real-Mode 191
Records 45; 151
  Aufbau 152
  variante 151; 156
Rekursion 201
repeat-Schleife 116
repeat-until 116
Reset 169
Reset() 164
ReWrite 169
Routinen zur Verwendung
  von Strings 65
Rückgabetyp von Funktionen 196
RunError 121; 125

S

Schleifen 96; 100
  Funktionsweise 114
  fußgesteuerte 101
  kopfgesteuerte 101
Schleifenausführung vorzeitig
  abbrechen 123
Schleifenkörper 100
Schleifensteuerung 123
Schlüsselworte 24
Schreib-/Lesezeiger 166
Seek() 173
Segment 223
Semikolon 75
Sequenz 96
Set 45
shl 74; 85
Shortcurcuit Evaluation 83
shortint 61
shr 74; 85

single ... 63
SizeOf() ... 237
Softwarebibliotheken ... 186
Sonderzeichen in Pascal ... 23
Spaghetticode ... 120
Speicher, verlorener ... 227
Speicheradressen ... 233
Speicherplätze, variable ... 41
Speicherprobleme ... 226
Speicherverwaltung, dyn. ... 45; 236
Speicherzugriffe, illegale ... 226
Sprünge ... 120
Sprunglabel ... 121
Stack ... 225
Stackwerte ... 225
Standardanweisung ... 25
Standardbibliothek ... 21
Standarddatentypen ... 42
Standardeingabekanal ... 69
Standardunits ... 190
Steuerwert der varianten Records 156
Str ... 67; 92
String ... 52; 66; 190
  Zeichenkette ... 24
Stringfunktionen ... 66
Stringkonstante ... 34
StrPas ... 92
StrPCopy ... 92
Struktogramme ... 96
Struktogrammeditoren ... 103
Struktur
  in Strukturen ... 155
  Platzverbrauch ... 154
  und Array ... 156
  Zugriff ... 154
Struktur.Element ... 156
Strukturelemente ... 152
Subroutinen ... 186
  Definition ... 192
  Forward-Deklaration ... 194
  Arten ... 187
  Aufruf ... 200
  Entwurf ... 188
  Funktionen ... 187
  öffentliche und private ... 218
  Prozeduren ... 188
  und Variablen ... 203
  verschachtelte Definitionen ... 198
  Vorabdeklaration ... 194
Succ ... 55
Syntax ... 22
  Diagramme ... 22
  rekursive Diagramme ... 22
Syntaxcheck ... 21
System ... 190

T

Text ... 163
Textausgabe ... 35
Textdateien ... 163
  typisierte ... 167
  Zugriff auf ... 163
Textkonstante ... 48
to ... 118
TPW.TPL ... 191
True ... 43; 47
True-Zweig ... 104
Trunc ... 92
TURBO.TPL ... 191
Turbo3 ... 190
Typbeschreibung ... 129
Typdefinitionen ... 128
  explizite ... 131
Typdeklarationen
  explizite ... 128
  implizite ... 128
type ... 128; 131; 152
Typecast ... 93
Typen umbenennen ... 133
Typkonstanten ... 135
Typumwandlung ... 90
  erzwungene ... 91
Typvereinbarungen
  globale ... 210
  lokale ... 209

U

Überlauf, Ganzzahl- ... 59
Umwandlungsfunktionen ... 92
UND bitweise ... 86
Unit ... 21; 186; 215
  Aufbau ... 215
  Definition ... 215
  Interfaceteil ... 216

(Unit)
String ... 65
überlagerte ... 218
Unterbereiche ... 138
uses ... 32; 189
in Units ... 217

V

Val ... 67; 92
var ... 51; 204
var in menge ... 142
Variablen ... 40
deklarieren ... 51
globale ... 203; 211
Initialisierung ... 214
lokale ... 203; 212
Wertzuweisung ... 53
Variablenvereinbarung ... 51
globale und lokale ... 211
Variante Records ... 156
Vergleiche bei Zeigern ... 233
Vergleichsoperatoren ... 79
Verzweigung ... 104; 96
Struktogramm ... 98
verschachteln ... 107
Vorabdeklarationen
von Subroutinen ... 194

W

Wahrheitswerte ... 47; 62
while-Schleife ... 59; 114
Whitespaces ... 24
WinAPI ... 191
WinCrt ... 33; 191
WinDOS ... 190
WinPrn ... 191
WinProcs ... 191
WinTypes ... 191
Wirth ... 20
with ... 154
word ... 61
wordbool ... 62
Write ... 169
Write() ... 34
WriteLn() ... 33; 34

X

xor ... 74

Z

Zahlen
einlesen ... 69
ausgeben ... 35
Zahlenkreis ... 60
Zählschleife ... 102
mit for ... 118
Zeichenketten ... 44; 64
erlaubte Zeichen ... 24
Zeichenkettenaddition ... 78
Zeichenkettenkonstante ... 34
Zeichenkettenvergleiche ... 80
Zeichenkonstante ... 47
Zeiger ... 45
auf Zeiger ... 230
typisierte und nicht typisierte . 227
Zeigerdatentypen ... 229
Zeigerinhalte, Zugriff ... 232
Zeigerinitialisierungen ... 233
Zeigerketten ... 230
Zeigervariablen ... 223
Anlage ... 227; 228
Zugriff
auf Strukturen ... 154
auf variante Records ... 158
Zugriffsfehler
bei Arrays ... 149
Zuweisungen ... 53
Zweierkomplement ... 60

Programmierung

ECON COMPUTERTASCHENBÜCHER MACHEN DEN PC SO SYMPATHISCH

Ulrich Cuber
Heino Wenzel
Das Einmaleins der C-Programmierung

ECON TB 28058
400 Seiten, DM 36,90

Mit Symantec C-Compiler, Tools und Beispielen auf der beiliegenden CD-ROM

C hat wegen seiner prägnanten Schreibweise den Ruf, schwer erlernbar zu sein. Den Beweis des Gegenteils liefert dieses Buch. Sie finden in ihm alles, was Sie brauchen, um schnell in die Programmiersprache C einzusteigen:

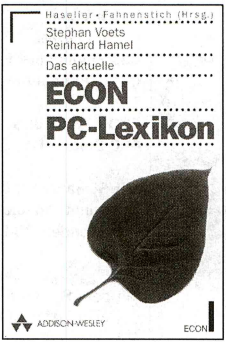
Zehn übersichtliche Kapitel mit überschaubaren Beispielprogrammen führen Sie in die Sprachelemente von C ein.

Mit dem Symantec C-Compiler, der auf der CD-ROM enthalten ist, können Sie alle Schritte sofort auf Ihrem PC nachvollziehen.
Außerdem enthält das Buch zahlreiche Tips zur richtigen Strukturierung eines Programms, um dessen Übersichtlichkeit und Wartbarkeit zu erhöhen. Sie lernen die wichtigsten C-Bibliotheksfunktionen kennen und erfahren, wie Sie eigene Funktionen erstellen können.
Die CD-ROM enthält außerdem ausgewählte C-Tools und Quelltexte, mit denen Sie nach den ersten Schritten Ihr C-Repertoire durch Selbststudium erweitern können.

ECON Taschenbuch-Verlag Postfach 300321 40403 Düsseldorf

Grundlagen

ECON COMPUTERTASCHENBÜCHER MACHEN DEN PC SO SYMPATHISCH



Stephan Voets
Reinhard Hamel
Das aktuelle ECON PC-Lexikon

ECON TB 28071
384 Seiten, DM 19,90

Das ECON PC-Lexikon betrachtet die Fachbegriffe aus der Sicht des Anwenders. Die Erklärungen richten sich nach den Erfahrungen und dem Wissensbedarf von PC-Anwendern, die z.B. schnell Informationen über einen ihnen unbekannten Begriff erhalten wollen, der z.B. in einer anwendungsorientierten Computerzeitschrift wie selbstverständlich verwendet wird.

Mehr als 2.000 Stichwörter zu Begriffen der Hard- und Software finden Sie in diesem PC-Lexikon, das sich durch eine hohe Aktualität auszeichnet. Mehr als 400 der aktuellen Begriffe werden Sie in anderen, vergleichbaren Lexika vergeblich suchen.

Eine Besonderheit des ECON-PC-Lexikons sind die zahlreichen Praxistips, die die Erklärungen ergänzen und besonders dem PC-Einsteiger die praktische Bedeutung der Begriffe nahebringen.

ECON Taschenbuch-Verlag Postfach 300321 40403 Düsseldorf

ECON Computer Taschenbücher
Das Original

Ulrich Cuber
ist nach seinem Studium der Chemie als freier EDV-Dozent tätig und hat mehrere PC-Bücher veröffentlicht.

Heino Wenzel
ist nach seinem Studium zum Höheren Lehramt nunmehr seit 10 Jahren in der EDV-Organisation und Ausbildung tätig. Gemeinsam betreiben sie eine eigene Beratungs- und Schulungsfirma.

Kategorie:
Programmierung

Schwierigkeitsgrad:

Originalausgabe

Das Einmaleins der Pascal-Programmierung

Pascal ist - gerade für Einsteiger - immer noch eine der beliebtesten Programmiersprachen, da die Pascal-Syntax sehr klar ist und viele Programmierfehler bereits beim Kompilieren entdeckt werden können. Dieses Buch führt Sie anhand von übersichtlichen Beispielprogrammen Schritt für Schritt in die Programmierung mit Pascal ein. Die Pascal-Sprachelemente, zu denen die Datentypen, die Operatoren und die Möglichkeiten der Ablaufsteuerung gehören, werden ausführlich erläutert.
Mit diesem Wissen ausgerüstet, sind Sie schnell in der Lage, auch eigene erste Programme zu schreiben.

Aus dem Inhalt:

- Datentypen und Verwendung von Variablen
- Operatoren und Ablaufsteuerung
- Modulare Programmierung mit Prozeduren und Funktionen
- Einsatz von Zeigern

mit allen Beispielen des Buches und mehr als 3 MByte weiteren Sourcen und Tools

ISB N 3-612-28086-4 DM +029,90

1

02990

Österreich S 221,– 9 783612 280862

ADDISON-WESLEY

ECON